NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AN ANALYSIS OF SPECWARE AND ITS USEFULNESS IN THE VERIFICATION OF HIGH ASSURANCE SYSTEMS

by

Daniel P. DeCloss

June 2006

Thesis Advisor:	Timothy Levin
Co-Advisor:	Cynthia Irvine

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** June 2006	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: An Analysis of Specware and its Usefulness in the Verification of High Assurance Systems			**5. FUNDING NUMBERS**
6. AUTHOR(S) DeCloss, Daniel P.			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

Formal verification is required for systems that require high assurance. Formal verification can require large and complex proofs that can drastically affect the development life cycle. Through the use of a verification system, such proofs can be managed and completed in an efficient manner. A verification system consists of a specification language that can express formal logic, and an automated theorem tool that can be used to verify theorems and conjectures within the specifications. One example of a verification system is Specware. This thesis presents an analysis of Specware against a set of evaluation criteria in order to determine the level of usefulness Specware can have in the verification of high assurance systems. This analysis revealed that Specware contains a powerful specification language capable of representing higher order logic in a simple and expressive manner. Specware is able to represent multiple levels of abstraction and generate proof obligations regarding specification correctness and interlevel mapping. The theorem prover associated with Specware was found to be lacking in capability. Through this analysis we found that Specware has great potential to be an excellent verification system given improvement upon the theorem prover and strengthening of weaknesses regarding linguistic components.

14. SUBJECT TERMS Verification, High Assurance Systems, Separation Kernel, Specware, Refinement, Theorem Prover, Information Assurance			**15. NUMBER OF PAGES** 111
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AN ANALYSIS OF SPECWARE AND ITS USEFULNESS IN THE VERIFICATION OF HIGH ASSURANCE SYSTEMS

Daniel P. DeCloss
Civilian, Naval Postgraduate School
B.S., Northwest Nazarene University, 2004

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2006

Author: Daniel P. DeCloss

Approved by: Timothy Levin
Thesis Advisor

Cynthia Irvine
Co-Advisor

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Formal verification is required for systems that require high assurance. Formal verification can require large and complex proofs that can drastically affect the development life cycle. Through the use of a verification system, such proofs can be managed and completed in an efficient manner. A verification system consists of a specification language that can express formal logic, and an automated theorem tool that can be used to verify theorems and conjectures within the specifications. One example of a verification system is Specware. This thesis presents an analysis of Specware against a set of evaluation criteria in order to determine the level of usefulness Specware can have in the verification of high assurance systems. This analysis revealed that Specware contains a powerful specification language capable of representing higher order logic in a simple and expressive manner. Specware is able to represent multiple levels of abstraction and generate proof obligations regarding specification correctness and interlevel mapping. The theorem prover associated with Specware was found to be lacking in capability. Through this analysis we found that Specware has great potential to be an excellent verification system given improvement upon the theorem prover and strengthening of weaknesses regarding linguistic components.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my advisors Timothy Levin and Cynthia Irvine for their continued support, patience, and guidance throughout this process. I would like to thank Dennis Volpano and George Dinolt who provided insight and direction during the research of this thesis. I would also like to thank Kestrel Technology, particularly Alessandro Coglio, for their support with Specware.

This material is based upon work supported by the National Science Foundation under Grant No. DUE0414102. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

High assurance computing and communication systems are evaluated to a high level of trust based in part on a formal verification that the actions of the system adhere to the established security policy. This is important due to stringent requirements on high assurance systems and their development process. The Common Criteria (CC) is used to evaluate, certify, and accredit systems and imposes requirements such that any system requiring a high level of trust (i.e. Evaluation Assurance Level 7 or EAL7), must undergo a rigorous life cycle including the use of formal verification of its security properties [Com06]. Examples include systems housing information at multiple classification levels, avionics software, missile guidance software, and even critical infrastructure management systems for water, power, and gas. All of these types of systems are required to be correct and must not contain errors or malicious artifacts that might result in the leak of sensitive information or the loss of human life. One way to ensure that the system is correct is to incorporate formal verification in the development life cycle.

Formal verification is thus a necessity for high assurance systems, but the level of effort associated with manual verification can be unreasonable due to large and complicated proofs. The use of an automated verification system can increase the efficiency and productivity of formal verification. There are several verification systems available and the choice between such systems is important and must be considered carefully based on the scope of project requirements.

In this thesis we analyzed Specware, a verification system developed by Kestrel Development Corporation, to determine the level of usefulness it could have in the verification of high assurance systems. We are evaluating Specware as a candidate for use on the Trusted Computing Exemplar project [Irv04]. For our analysis we adapted a set of evaluation criteria presented by Ubhayakar [Ubh03]. We conducted a simple experiment as a basis for evaluation. The experiment required familiarity with Specware and the capabilities of MetaSlang, Specware's specification language. The experiment consisted of creating a formal model with a basic security theorem based on a separation kernel security policy presented by Levin, Irvine, and Nguyen [Lev04]. Furthermore, we

created a formal top level specification as a refinement of the model and investigated the interlevel mapping capability within Specware. We continued to evaluate Specware through analysis of its ability to automatically generate theorems and conjectures at each specification level as well as conjectures associated with the interlevel mapping. Finally we analyzed Specware's theorem proving capabilities by attempting to prove the basic security theorem in the formal model and all of the conjectures associated with the model, formal top level specification (FLTS), and interlevel mapping.

This thesis presents our experimental findings and discusses the strengths and weaknesses of Specware corresponding to the adapted verification system evaluation criteria. We will present a brief overview of Specware and MetaSlang and its basic components. We will then describe the separation kernel formal model and FTLS developed in Specware and the technique used to produce the interlevel mapping. Finally, we will conclude with our analysis of Specware against the adapted evaluation criteria and present our conclusions and recommendations for future work. Overall, we found Specware to be a powerful tool with potential to be highly useful in the verification of high assurance systems; however, currently, a few aspects of the tool has weaknesses. Specware is under continued development and progress will hopefully be made in these areas.

II. BACKGROUND

A. FORMAL METHODS AND THEIR IMPORTANCE

The process of developing a high assurance system is naturally arduous. Yet the motivation to undergo such a process relies on the outcome provided. When a high assurance system is completed and implemented, one can be assured with a high degree of confidence that the system will behave correctly and appropriately. Appropriate behavior could be considered as system behavior that is intentional, free from malicious or inadvertent side affects. When discussing what types of systems should be developed using a high assurance methodology, many examples are immediately present, including but not limited to medical technology systems, aviation systems, and multilevel secure systems. Taking these examples one can immediately see how a small bug or glitch in the system could be disastrous, possibly resulting in the loss of human life.

The high assurance methodology ensures that a system will undergo a rigorous development life cycle in order to eliminate bugs and prove that the functionality is necessary and sufficient. The system must also be evaluated to determine its level of assurance. The evaluation process incorporates stringent guidelines relating to the development life cycle. The entire process is time consuming due to the guidelines that must be followed and heavy documentation associated with those guidelines. Such documentation provides a clear outline that developers and engineers can follow in order to verify system requirements and functionality. The documentation also serves as an invaluable reference for system maintenance. And finally the documentation is used for system evaluation. Although the process can seem almost overwhelming, it results in a system that can be verified to meet its specified requirements and desired functionality. This methodology is also vitally important when describing the security properties of the system.

Formal methods are the use of mathematics to prove certain properties about a system. Formal methods involve several levels of abstract descriptions of the system's security properties and desired functionality. Applying formal methods to the design and implementation of high assurance systems can be described through the following steps:

1. Security Policy
2. Security Model
3. Formal Top Level Specification
4. Implementation

For each step the appropriate verification must be achieved in order to maintain a correct mapping to the previous level, ending up back at the security policy. The goal is to sequentially refine the security policy to the implementation of the system such that the implementation is shown to be a valid representation (i.e. "maps to") the policy. The process can seem quite simple yet it is quite rigorous. It could take years to develop a high assurance system where the security policy is provably secure. Due to the expense in terms of time, money, and expertise, formal methods are mostly used in the development of systems that require trusted security properties to protect high valued information. One important reason formal methods are used is to provide a high level of confidence that the implementation meets the specification. Thus formal methods provide assurance that the security properties will be provided as specified. Another reason is that a formal security policy model provides developers with a single point of reference that defines exactly what is to be implemented. Thus formal methods provide an accountability mechanism for the developers and a solid reference framework to ensure that the security of the system can be understood. Landwehr described it well when stating that formal methods provide a concise organization of the complexity of "computer" and "security". Thus they provide a definition of what security actually means and how it can be determined with relation to the computer's behavior [Lan81]. Finally, formal methods provide the means to answer the question of whether the system is secure or not based on the proof of the basic security theorem.

B. THE REFINEMENT PROCESS IN FORMAL METHODS

As noted previously there are several levels of abstraction when applying formal methods to a system and security policy. The term *security policy* can be quite vague if not put within the proper context. Sterne distinguishes between the security policy objective, the organizational security policy, and the automated security policy [Ster91]. In terms of formal methods, we are concerned with the automated security policy which is an abstract view of the desired functionality and security properties that the system

must contain or address. It is free of implementation details and provides the reference for the entire formal methods process. The construction of the automated security policy is vital as it serves as the backbone for the entire process.

The next two phases are the construction of the formal security policy model (FSPM) and then the formal top level specification (FTLS). The model is a mathematically structured statement of the security policy. It is a logical representation of the security policy which basically takes the English language stated policy and formulates the mathematical equivalent. Additionally, the model must make a significant progression toward the actual implementation of code. The model serves as the first stepping stone in the process and is a high level abstraction between the security policy and the implementation of the system. Keep in mind that the ultimate goal of this process is to ensure that the actual code behaves in no way violates the policy, and yet still contains the desired functionality. Thus a system full of NOPs is not a violation of the policy, but provides no useful functionality. The code preserves the security properties and is based ultimately from the model. Thus the model must be an accurate representation of the policy in order to maintain a high level of assurance. The model consists of two major components. The first is a general model of a system plus a set of operations, and the second is a definition of security that constrains the system. Constraints are stated in the form of axioms and conjectures, which must be proven based on the constraints. Ultimately, the basic security theorem must be proven true based on all the constraints put on the system. Thus a secure system is defined as one in which all constraints are satisfied [Ubh03].

The FTLS is the second level of abstraction from the policy and steps towards the implementation in terms of specificity. It defines all interfaces with appropriate parameters. It represents all inputs and outputs necessary for the system and also describes the exceptions and effects that processing will have on the state of the system. . The FTLS describes all actions that the system takes and the impact that those actions will have on the security properties of the system. The formal nature of the FTLS allows for proof that it maps to the model and transitively supports the security policy.

The FTLS must support three main goals. First, it must support a proof that the system design enforces the security policy. Secondly, it must provide a basis for an analysis and catalogue of all covert storage channels. Lastly, it must provide a criterion of correctness for the implementation [NSA87]. The major difference between the security policy model and the FTLS is the level of specificity. The important distinction is that the FTLS represents a significant progression from the policy to the implementation. This sequential progression must be provably secure in that each refinement is shown to map to the previous level of abstraction and ultimately the policy. The sequential refinements can be quite challenging and the proofs can become cumbersome. Performing the proofs of the model and FTLS by hand could consume a large amount of time and human resources. Thus the need for tools that aid in the refinement and representation of security models and policies is quite evident.

C. VERIFICATION SYSTEMS

The use of tools that improve the efficiency and correctness of the verification process is necessary to produce a secure system. Tools that can assist in the verification of high assurance systems include formal specification languages and theorem provers. Languages provide a means to represent models and policies and to express the refinement of such models in a formal manner. Theorem proving tools help to minimize the manual effort required to arrive at a valid proof. Theorem provers can either be interactive or automatic. An interactive prover requires the user to initiate proof commands to guide the system through the verification, whereas an automatic prover attempts to reach a proof without any guidance or involvement from the user except at the invocation of the prover. Essentially a theorem prover processes specifications and determines if the conjectures are correct and valid. A specification language can be used to specify a system and to declare conjectures and proof obligations. We will briefly describe three tools that are in use to aid in the verification of high assurance systems.

1. Ina Jo – Category Theoretic

Ina Jo is the specification language processor included in the Paramax Formal Development Methodology (FDM) software [Par92]. The Ina Jo processor reads specs

that are written in the Ina Jo specification language and automatically generates correctness conjectures. The FDM tool set includes two theorem provers, the Interactive Theorem Prover (ITP), and the Natural deduction Automated Theorem proving Environment (Nate). Ina Jo is derived from first order logic with quantification. An Ina Jo specification describes system states, state transitions, and correctness criteria. Ina Jo also provides linguistic elements to describe multiple levels of abstraction as well as the mapping from one level to another. The conjectures must then be proven and once they have been proven they become theorems. Ina Jo theorems fall into three categories:

- Initial condition theorems state that the initial states satisfy the correctness criteria.
- Transform theorems state that transforms preserve the correctness criteria.
- Mapping theorems state that a lower level spec properly implements its parent.

Once Ina Jo has generated the conjectures the previously mentioned theorem prover is used to verify them. Some nice features that Ina Jo provides include a precise way to state what level the specification represents through the use of the LEVEL statement. Thus a declaration of the spec could appear as *LEVEL model* and then *LEVEL ftls UNDER model* [Par92].

2. PVS – Type Theoretic

The Prototype Verification System (PVS) is a verification system that provides an interactive specification environment that supports writing formal models and specifications and theorem proving. PVS provides an all inclusive environment that contains its own powerful specification language and interactive theorem prover [Ubh03]. Certain low level proof steps are automatically included in PVS, but the user must initiate the higher level steps to create goals and subgoals that need to be proven in order for the specification to be correct.

3. Specware – Category Theoretic

Specware is a utility created by Kestrel Institute which provides a specification language, MetaSlang, and the ability for refinement of specifications to produce code in a target programming language. MetaSlang provides linguistic elements to describe multiple levels of refinement and its processor generates the associated proof obligations. Specware incorporates the theorem prover SNARK developed by SRI [Kes04]. Specware comprises multiple specs and refinements of specs to ultimately produce provably correct code. Refinement is conducted through the use of morphisms. Morphisms are a concept based from category theory which are defined by McDonald and Anton as truth preserving mappings of one spec into another [McD01]. Thus the two major stages in producing a Specware application include building the spec and then refining the spec [Kes04]. It is the intent of the rest of this document to analyze Specware and determine the degree to which it is useful in the verification of high assurance systems. A more extensive overview of Specware is provided in Chapter III and Chapter IV provides an analysis of Specware's application to the verification paradigm.

D. THE VERIFICATION PARADIGM

Due to the extensive nature of developing a high assurance system, it is important to choose a verification tool, or set of tools, that will be useful throughout the development process. Ubhayakar presented a set of evaluation criteria for verification tools [Ubh03]. Ideally, the verification tool will support formal specifications, proofs, refinement and covert channel analysis, and provide adequate documentation of the same.

When determining a tool's usefulness in the verification of high assurance systems, we desire to evaluate it based on a set of objective criteria in order to show its relative effectiveness. The analysis is performed by developing specification models and proofs based on a security policy in the tools' specification language. In terms of the verification of high assurance systems, we are mainly concerned with the tools' usefulness in developing the security policy model, FTLS, and the proofs associated with the mapping. Naturally this type of analysis will depend on many factors that might exist beyond the initial set of evaluation criteria. Such dependencies are very important and

should not be overlooked, for instance, suppose that a tool is found to be quite useful but requires expensive training costs. The decision to use the tool must be made according to the available resources and development schedule. Thus, it is important to state ahead of time, what criteria beyond the initial set should be considered when performing the analysis of the tool. Ubhayakar [Ubh03] presented an initial table of evaluation criteria, which we have extended as seen in Table 1:

Evaluation Criteria	***Definition***	***Utility***
Product Maturity	*A tool should be old enough and currently maintained and supported*	*Specific questions need to be answered in a timely manner regarding syntax and specification language*
Usability of Tool and Verification Environment	*The level of simplicity and flexibility of operations provided to the user*	*The interface and commands should be simple to understand and should provide syntax highlighting and error checking to increase efficiency*
Theorem Proving	*Interactive versus automated theorem proving*	*Theorem proving should be easily integrated and provide meaningful descriptions of errors and logging capabilities*
Specification Language	*Syntactical elements of the language*	*Learning curve associated with language should be minimal to provide efficient generation of specficiations*

Executable Specifications	*Ability to test system directly from specification language*	*Executable specifications provide the user with a general "feel" for the system*
Multiple Levels of Abstraction	*Refinement capabilities from more abstract specifications to more concrete specifications*	*Multiple levels of abstraction provides ability to verify that the top level specification satisfies security policy*
Automatic Generation of Conjectures	*Ability to automatically state items which must be proven*	*This aids in ensuring that all obligations regarding the system are being addressed*
Semantics	*Powerful expression of logic with minimal complexity*	*Underlying logic and foundational theory affects the expressiveness of the tool regarding system properties*

Table 1. Specware Evaluation Criteria

For this thesis, we will analyze Specware and determine its usefulness in the verification of high assurance systems. We will develop formal specifications in Specware based upon a simple separation kernel security policy. We will then analyze the specifications in order to describe the utility of Specware regarding the verification paradigm. This analysis is not to determine Specware's usefulness in the general sense, but to describe the level of its usefulness when developing a formal security policy model and FTLS and its proving capabilities. The next chapter will provide an overview of Specware and describe its history as well as some projects that it has been used on. Following the discussion of Specware, we will present our experiment and analysis.

III. SPECWARE OVERVIEW

A. SPECWARE DESCRIPTION

Specware was developed and is supported by Kestrel Development Corporation and has been in production for over a decade. The version of Specware used in this project is version 4.1.3. The philosophy behind Specware is to provide an automated tool to aid in a refinement-based approach to formal software development. Formal software development implies the rigorous construction of executable code that meets a well-defined specification [McD01]. Specware's refinement process is based on the mathematical foundation of category theory, which is concerned with the manner in which properties are preserved between different objects. In category theory, morphisms are the relations between objects [Sri96]. The advantage of category theory as the foundation of Specware is that it enables the production of a well-defined stepwise refinement from an abstract specification to concrete implementation. Specification morphisms preserve the structure of one specification through the translation to another specification and preserve theorems across the specifications [Sri95]. Thus refinement capabilities in Specware provide a logic-preserving process wherein each refinement can be proven to preserve the properties of the more abstract specification [McD01]. The entire goal of Specware is to provide a framework to produce provably correct code and aid in the development of efficient, high-assurance software [Pav03]. Based on the description of the verification paradigm and formal methods process, Specware's foundation is appealing for developing high assurance systems.

B. SPECWARE FUNCTIONALITY

1. MetaSlang

Specware is a tool to build and refine specifications, generate code from specifications, and prove properties regarding those specifications and refinements. The specification language used in Specware is called MetaSlang. The *Specware Language Manual* contains a detailed description of the MetaSlang grammar, including a BNF description. MetaSlang includes syntactic constituents for describing functional semantics within a specificaiton as well as constructs for describing composition,

refinement, code generation, and proof capabilities. Specification constituents include types, expressions, and axioms which can be used to describe domain-specific formalisms [Kes04]. The MetaSlang grammar follows a functional style of programming, which is valuable for proving properties regarding functions; however, the functional style causes issues when trying to represent state, which is discussed in Chapter V, Section E and future work. The basics of MetaSlang are briefly described in this section, but the reader is recommended to refer to the Specware documentation for a more comprehensive explanation.

a. Specs

"A specification is a finite presentation of a theory in higher-order logic" [Sri95]. Specifications, or *specs*, provide the means to describe abstract concepts of the problem domain. *Specs* contain types for describing collections of values and operations, or functions on those values. *Specs* also contain axioms and definitions which define the actions and properties of types and operations. A *spec* can be extended by importing other *specs*. This copies the imported *spec* into the target *spec* creating a larger and potentially more complex *spec*. *Specs* are also the objects used in morphisms which define the part-of or is-a relationship between two *specs*. Morphisms allow for refinement of *specs* and provide the utility to take simple abstract specifications, and refine them to more concrete, complex specifications [Kes04]. The general form of a *spec* definition in MetaSlang is a sequence of one or more declarations, as shown in Figure 1.

```
spec example_spec
   {declaration}
   ...
endspec
```

Figure 1. Spec Definition

[1] Reserved words in MetaSlang will appear in bold throughout all figures in this document.

b. Types

Types are collections, or sets, of objects and expressions that characterize those objects. Specware has several inbuilt types provided in its libraries which are imported automatically for every *spec* processed by Specware. Specware's libraries continue to grow as time goes on and when creating specifications it is important to consider if they can be reused across multiple problem domains. Some example type declarations are shown in Figure 2. Notice in the definition of *Mode* the vertical bar precedes each element.

```
type Resource

type String

type Mode = | READ | WRITE | EXECUTE
```

Figure 2. Type Declarations

c. Ops and Defs

An operation, or *op* in MetaSlang, is used to describe instantiations of types. *Ops* are used to declare explicit types as well as declare functions that will perform an operation based on the types given in the declaration. Figure 3 shows example of *op* declarations. *Ops* can be monomorphic (i.e. strict typing) as seen by the definition of *Name* which can only be of type *String*. *My_Predicate* is also an example of a monomorphic *op* that can only take a *String* as input and will only result in a *Boolean* value. *Ops* can be polymorphic, as seen in *My_Function*, indicating that the *op* can be used across different types. Thus *My_Function* takes two parameters of different or the same type and returns a value of a third type. It is clear that the declaration of a polymorphic *op* describes little context for its use, but the definition of the *op* will bring clarity to its context and proper use.

```
    op Name : String
  %Polymorphic op
    op My_Function: [a,b,c] a * b -> c
    op My_Predicate: String -> Boolean
```

Figure 3. *Op* Declarations

Once the operations have been declared, definitions, or *defs*, of *ops* are used to define the behavior and constraints (i.e. the semantics) of the *ops*. Thus an *op* definition corresponds to a previously declared *op* and must correspond to the signature of the *op* declaration. An *op* definition is considered a special notation for an axiom and is expresses the same logic that an axiom might express; however, a *def* might still have proof obligations associated with it, whereas an axiom is automatically assumed to be true and has no obligations. Thus, it is encouraged to use *defs* as much as possible in order to be as precise as possible [Kes04]. A *def* can also be used to declare constants. Figure 4 shows the use of *defs* to declare a constant *Limit* and the definition of *op f*.

```
  def Limit = 12
  %Declaration
    op f : Nat -> Nat
  %Definition
    def f(n) = 3*n
```

Figure 4. *Op* Definitions

d. Claims: Axioms, Conjectures, and Theorems

Axioms, conjectures, and theorems are all considered types of claims within Specware. All claims must be of type Boolean. Conjectures and theorems are claims that must be proven through the use of *op* definitions and axioms. Specware will automatically generate conjectures based on *op* declarations, but the user can also create conjectures as well [Kes04]. Currently, conjectures and theorems are synonymous in the way that the Specware processor handles the two types of claims, but as Specware becomes interoperable with other theorem provers, a difference might be distinguishable. Some example claim definitions are:

```
axiom Example_1 is fa (x: Integer, y: Integer)
     f(x) = f(y) => x = y
conjecture Example_2 is fa (w: Integer, z: Integer)
     (z*w = 0) => (z =0 || w = 0)
theorem Example_3  is fa (a: System_Transform)
     Transform_Secure(a)
```

Figure 5. Claim Definitions

2. Refinement and Morphisms

The goal of refinement is to take an abstract description of a solution and develop a more precise description which can be shown to be a correct representation of the initial description. The process of stepwise refinement provides a sequential composition of refinements where each refinement introduces new detail and is shown to preserve all previous properties [Sri95]. The refinement process in Specware consists of an initial specification that expresses the high level requirements and then continues with refinement *specs* that indicate design and implementation decisions. Thus the stepwise refinement of specifications proves the existence of a valid implementation of the initial specification [Pav03]. The glue that connects each pair of refinement *specs* is the specification morphism.

In Specware the morphism from one specification (source) to another (target) is a property and structure-preserving mapping such that every type and *op* in the source *spec* is directly mapped to a type and *op* in the target *spec*. The morphism allows us to speak of items (types, *ops*, axioms, and *defs*) in the target spec as images of items in the source spec. The images of the axioms and definitions in the source *spec* are conjectures generated by Specware to be proven in the target *spec*. Thus the morphism shows that all properties of the source *spec* are satisfied by the target *spec*. Thus each level of refinement is shown to satisfy the conditions from the level above, providing a proof chain from the most concrete refinement to the initial specification [Kes04].

A morphism between two specifications is declared by indicating the source spec mapping to the target *spec* with a specialized arrow (i.e. +->) in between. Specware will automatically map types and *ops* in the source *spec* to types and *ops* of the same name in the target *spec*. If type names differ between *specs*, then the mapping between types must be made explicitly. Every type and *op* in the source specification must map to another type and *op* in the target. The syntactic elements of the morphism include the # symbol which is used to identify the particular *spec* within the file, and the +-> symbol which is the mapping symbol used to express individual element mapping. An example morphism declaration is seen in Figure 6.

```
Sample_Morphism =
  morphism Source_Filename#Source_Spec ->
    Target_Spec {
       source_type1 +-> target_type1,
       source_op1 +-> target_op1}
```

Figure 6. Sample Morphism Declaration

3. Proof Obligations

Proof obligations are properties regarding relationships of items within a specification and must be shown to be true in order for the specification to be correct.

Specware automatically generates proof obligations for definitions and the user can also state explicit obligations in the form of conjectures or theorems. Proof obligations are also automatically generated for a morphism. Proof obligations that are automatically generated within *specs* are typically related to type checking and *op* definitions. Automatically generated obligations do not appear within the original *spec*, and in order for the *spec* to be proven true, the user must invoke them by using ***obligations*** command. This command can be present in the definition of a unit, as seen in Figure 7 or it can be given from the Specware shell in combination with the ***show*** command. We will discuss the Specware shell environment in the next section.

Once the obligations have been invoked, they can then be proved using the ***prove*** command [Kes04]. Obligations must be proved one at a time, but not necessarily sequentially. For organization purposes, we found it beneficial to maintain a separate file containing only proof units which assigns proof obligations unique identifiers. This allowed us to select which proofs should be attempted in a proving session as opposed to attempting all proofs in every proving session. Figure 7 is an example of a file that assigns proof obligations from a given *spec* to a unit and it also demonstrates assigning individual proof obligations from the same *spec* to proof units. Note that a unit references a label to an assigned element in Specware (e.g. *p1* is a unit).

```
spec_obligations = obligations File#Sample_Spec

p1 = prove obligation1 in File#spec_obligations

p2 = prove obligation2 in Spec#spec_obligations
```

Figure 7. Proof and Obligation Declarations

This technique allows for unambiguous identification of obligations and provides a reference when analyzing the log files associated with each proof attempt. If the theorem prover is not able to prove the claims, this does not mean that the proof does not exist as the theorem prover may not be smart enough to figure it out. If a proof fails, the user can walk through the proof by hand and determine if a solution exists or if the

specification needs modification. Obligations associated with a morphism can be seen using the ***show obligations*** command within the Specware shell. The Specware processor will generate a separate *spec* which will contain all obligations necessary to satisfy the morphism [Kes04]. Figure 8 shows the way to view our *Sample_Morphism* oblgations from within the Specware shell.

```
show obligations Filename#Sample_Morphism
```

Figure 8. Morphism Obligations

4. Specware Shell

The processing of Specware specifications is performed within the Specware shell. The Specware shell is a command line environment. The Specware distribution package comes with XEmacs which can run the Specware shell, but the Specware shell can be run outside of XEmacs. XEmacs provides some features, such as syntax highlighting, that are useful for *spec* development in MetaSlang. The Specware shell contains several commands including basic file system operations such as ***cd*** and ***dir*** but also commands specific to processing Specware units such as the ***proc*** and ***show*** commands. The ***show*** command can be used to display the contents of units or proof obligations. Within the Specware shell, the user can create and process *specs*, generate proof obligations, send obligations to a theorem prover, and even evaluate constructive MetaSlang expressions [Kes04]. Readers are encouraged to review the Specware documentation to become more familiar with the Specware shell and development environment.

C. SUMMARY

In summary, Specware is a tool intended to aid in the process of formal software development through the use of stepwise refinement. The mathematical foundation of Specware refinement is category theory which provides a mathematical foundation for describing the relationships between objects and operations. This foundation allows us to describe and prove the relationship between specifications. Specifications are written in

MetaSlang. *Specs* contain *types*, *ops*, and *claims* (e.g. axioms, conjectures, and theorems) which formally represent the logic of a specific problem domain. The concept of stepwise refinement is achieved through the use of morphisms. A morphism is a relationship between specifications that describes how the properties of one map to the properties of another. All conditions of the source *spec* must be satisfied in the target *spec* in order for the morphism to be proper. Thus final refinement specification is shown to preserve the properties from the abstract specification.

Next we will analyze how Specware and its refinement features support the verification of high assurance systems. We will describe the development of a formal model in Specware based on a separation kernel security policy. Then we will describe the development of an FTLS in Specware as a refinement of the model. We will use the morphism feature in Specware to achieve the interlevel mapping and thus demonstrate refinement, which is required by the formal methods process.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. SPECWARE AND THE VERIFICATION PARADIGM

A. SEPARATION KERNEL OVERVIEW

To conduct our analysis of Specware within the verification paradigm we chose to create a policy model and FTLS of a separation kernel. A separation kernel provides a partitioning of all system resources under its control into blocks such that actions taken by active entities within any particular block are isolated and undetected by entities in other blocks. A separation kernel achieves this partitioning and isolation of entities through management and virtualization of shared resources such that each block is assigned a resource set over which it believes itself to have complete control. The only manner in which a block might communicate with another block is if a means for communication has been established explicitly. Such information flow properties are desirable in environments where certain flows are allowed based upon a flow policy. One example might be a Multi-Level Secure (MLS) system that manages a flow policy between different classification levels of data. Levin, Irvine, and Nguyen defined a model for a static separation kernel which provides least privilege information flow [Lev04]. For a comprehensive understanding of this model we recommend referencing the paper, but we will provide an overview of the model and discuss its specification within Specware.

The least privilege separation kernel model consists of a set of resources, a set of operations, a set of modes of flow (i.e. Read, Write, Read & Write), a distinct partitioning of the resources into a set of blocks, a block-to-block flow function, and a subject-to-resource flow function. The set of resources is composed of internal resources, i.e. those which are only available to the kernel, and exported resources to which an explicit reference is possible via the separation kernel interface [Lev04]. The set of resources is partitioned into blocks, where every resource belongs to one and only one block. Subjects are a subset of exported resources which represent the active entities of the system, such as processes, programs, etc. Subjects can invoke certain modes of flow with respect to other exported resources. The notion of this flow is called an effect, which consists of a subject, resource, and mode of flow. Note that the resource can be another subject, the only stipulation being that within an effect, the resource, or passive entity, is

an exported resource. The set of all possible effects is the cross product of the set of subjects, exported resources, and modes of flow. Next the model describes the notion of the flow policy. The flow policy dictates what types of flows are allowed between blocks, and what types of flows are allowed between subjects and resources. The block-to-block flow function defines the set of allowed flows between blocks. The subject-to-resource flow function defines the set of allowed flows between subjects and exported resources. Thus the subject-to-resource flow function and the block-to-block function together express the flow policy.

An operation is associated with a set of effects. For example, if the separation kernel includes a read operation, there might be several effects associated with that read depending on the implementation of the operation. Thus all operations possess a set of effects. The notion of a secure operation is defined as an operation in which all of its effects are considered secure. A secure effect is one in which the given flow between the subject and resource is allowed by the policy, as well as the flow between the blocks in which the subject and resource reside is allowed by the policy. Finally a secure system is one in which all of its operations are secure [Lev04]. The paper also goes on to describe the notion of partial ordering of blocks and a trusted partial ordering using trusted subjects; but for this work we did not implement the trusted partial ordering and refer the reader to the paper for a more comprehensive understanding of this aspect. Next we will describe our specification of the model within Specware.

B. DESCRIPTION OF SEPARATION KERNEL MODEL IN SPECWARE

1. Type Declarations

As described earlier, once the security policy has been clearly defined, the next step is to represent the policy in a formal model. The model states the essence of the policy in a basic security theorem, which must be proved in order to verify that the model is consistent with the policy. In this section we describe the specification of the separation kernel model written in Specware's Metaslang. The complete specification for the separation kernel model is given in Appendix A. First we declare a type called *Resource* which indicates the set of all resources available to the kernel. We then proceed to define a subtype *Exported_Resource* which indicates all resources which are

not internal to the kernel. Figure 9 shows the declaration of *Resource* and *Exported_Resource*.

```
type Resource
  op exported? : Resource -> Boolean
type Exported_Resource = (Resource | exported?)
```

Figure 9. Resource and Exported_Resource type declarations

In Specware we declare *Exported_Resource* as a subtype using a predicate that satisfies some condition indicating that it is an exported resource. This condition is left abstract and does not need to be defined in the model. Next we define subtype of *Exported_Resource* called a *Subject*. We declare *Subject* as a subtype in a very similar fashion as we defined the subtype *Exported_Resource* by using a predicate that must be true in order for it to be a subject. The definition of the subject predicate is left abstract in the model, but in the FTLS we refine the definition of the predicate. Notice that constructing subtypes in this manner provides a proper containment of elements such that *Subject* is a subset of *Exported_Resource* which is a subset of *Resource.* Figure 10 shows the declaration of subtype *Subject.*

```
  op subject? : Exported_Resource -> Boolean
type Subject = (Exported_Resource | subject?)
```

Figure 10. Subject type declaration

Following the *Subject* declaration we complete the type declarations by declaring the *Block, Mode, Effect,* and *Operation* types. The *Block* declaration utilizes the Sum type feature in Specware, which allows a partitioning of the type being declared. Thus for our *Block* declaration we declare names for blocks with which resources will later be associated. We use the terms *High, Medium,* and *Low* to represent how separation kernels are sometimes used, but these are merely labels and will have no semantic designation. Type *Mode* indicates the modes of flow that are taking place in the system and is declared in a similar fashion to *Block* such that the only modes of flow are *RD (for Read), WR (for Write), RW (for Read/Write),* and *NULL.* The declaration of the *Effect* type uses what is known in Specware as a record type where each effect consists of a *subject* which is of type *Subject*, a *resource* which is of type *Exported_Resource,* and a *flow* which is of type *Mode*. The final type declaration is an *Operation* which consists of a List of effects, or all the effects that are associated with each operation. Figure 11 shows the final type declarations.

```
type Block = | High | Medium | Low
type Mode = | RD | WT | RW | NULL
type Effect = {subject: Subject,
               resource: Exported_Resource,
               flow: Mode}
type Operation = List Effect
```

Figure 11. Block, Mode, Effect, and Operation type declarations

2. BB, SR, and Partition Function Declarations

Following the type declarations we declare functions that allow us to express the allocation of the resources to blocks as well as determine what types of flows are allowed between blocks and what types of flows are allowed between subjects and resources.

First we declare a polymorphic predicate that is used to indicate which entity is the active entity for both policies. This predicate is given the name *active?*, where the general convention for predicates is to end their names with a question mark signifying that it is a Boolean expression.

Next we declare the *BB* function which represents the block-to-block flow policy. This function takes two blocks, b_1 and b_2, where b_1 is the block of the active entity that causes the flow. The function returns the list of modes of flow that subjects in b_1 are allowed to perform on resources in b_2. We make this distinction in order for the model to be able to express a policy which allows, for example, the flow [*b1, b2, RD*], but does not allow the flow [*b2, b1, WR*]. In this case, the direction of the flow is the same (information is flowing from *b2* to *b1*), however, the cause of the flow is different. The *SR* function represents the subject-to-resource flow policy and is declared in a similar fashion as the *BB* function. The *SR* function takes a subject and exported resource as parameters, where the subject is the active entity, and returns a list of modes flow that the subject is allowed to perform on the exported resource. Note that this policy definition allows flows between two subjects, since subjects are defined as exported resources, which is why we declare the *Subject* to be the active entity.

Finally we declare the *Partition* function which takes an exported resource as input and returns the block in which it resides. In Specware, when a function is declared it is naturally assumed to be well-define and no constraint is needed to discuss its totality. Thus the *Partition* function is total, such that every exported resource is assigned to exactly one block, but multiple resources could map to the same block. Note that we need not define how the policy relations are populated. This is a convenient abstraction leaving the details of the initialization of these policies as a refinement. Figure 12 shows the declarations of these functions.

```
op active? : [a] a -> Boolean
op BB : {(b1,b2): Block*Block |
```

```
                    active? (b1)} -> List Mode
  op SR : {(s1,r2): Subject*Exported_Resource |
                    active? (s1)} -> List Mode
  op Partition : Exported_Resource -> Block
```

Figure 12. active?, BB, SR, and Partition Function Declarations

3. Policy Description Functions – SecureEffect and SecureOP

Now that we have described the types and functions that exist within the system, we need to express certain qualities about the security of the system. We do this by defining what it means for an effect and an operation to be secure. As mentioned previously a secure effect is an effect in which the flow is allowed based on the subject-to-resource and the block-to-block flow policies. Thus we can declare a function called *SecureEffect* which returns true if the effect is in fact secure. The definition of *SecureEffect* states that either the flow is *NULL*, which means that the subject will perform no action on the resource, or the flow is allowed by the *BB* and *SR* functions. Once we have the notion of a secure effect we can describe a secure operation in which all effects associated with the operation are secure. *SecureOP* is defined as an iterative search through the list of effects associated with the operation. The iteration clause states that if the operation consists of a head element, *hd*, and a tail, *tl*, which is another list of elements, then continue the process through the list by checking the head and recursively processing the tail. If all of the effects are found to be secure, then the entire operation is considered to be secure and the function will return true. Figure 13 shows the definition of *SecureEffect* and *SecureOP*.

```
%Policy Description
  op SecureEffect : Effect -> Boolean
```

```
def SecureEffect (effect) =
   (effect.flow = NULL ||
   (member(effect.flow, BB(Partition(effect.subject),
                           Partition(effect.resource)))
    &&
    member(effect.flow, SR(effect.subject,
                           effect.resource))))

 op SecureOP : Operation -> Boolean
def SecureOP (operation) = case operation of
                             | nil -> true
                             | Cons(hd, tl) ->
                                (SecureEffect(hd) &&
                                 SecureOP(tl))
```

Figure 13. SecureEffect and SecureOP definitions

4. Model Axiom and Basic Security Theorem

Before we can state the basic security theorem we need to include an axiom to support the basic security theorem. The axiom *operations* states that for all effects and operations, if an effect is a member of an operation, then its flow is either *RD, WT,* or *RW* and the flow is allowed by the *BB* and *SR* policies. Essentially this implies that all effects in an operation are secure, which implies that the operation is secure. In our initial development of the model we defined three operations that met the same properties as this axiom; however, we encountered mapping problems with our initial approach and resorted to stating this axiom. The mapping problem we encountered regarded the fact that we could not map multiple operations in the FTLS to only one operation in the

model. We will discuss this mapping problem further in Chapter V, Section C-6. It is important to remember the *operations* axiom because it will become a conjecture in the FTLS which must be proved based on our definitions of the operations.

Finally we can state the theorem which must be proved in order to ensure that the system is secure. The theorem plainly states that in order for the system to be secure, all operations must be secure. This is proved using the definitions and axioms we have already described. The basic security theorem does prove within Specware using Snark. Figure 14 shows the declarations of the axioms and security theorem.

```
%Axiom
    axiom operations is
      fa(e: Effect, o: Operation)
        member(e,o) =>
         (e.flow = RD &&
          member(e.flow, BB(Partition(e.subject),
                            Partition(e.resource))) &&
          member(e.flow, SR(e.subject, e.resource)))
         ||
         (e.flow = WT &&
          member(e.flow, BB(Partition(e.subject),
                            Partition(e.resource))) &&
          member(e.flow, SR(e.subject, e.resource)))
         ||
         (e.flow = RW &&
          member(e.flow, BB(Partition(e.subject),
```

```
                                Partition(e.resource))) &&
            member(e.flow, SR(e.subject, e.resource)))


%Theorem
  theorem Secure is
          fa(o: Operation) SecureOP(o)
```

Figure 14. Model Axioms and Basic Security Theorem

It can be seen that the model for the separation kernel security policy is fairly concise and yet provides enough detail to accurately express the security policy. The model is a significant progression towards the implementation and can now be refined through morphisms in Specware. Since the security theorem has been proved in the model, if we can prove that the FTLS satisfies the morphism theorems, then it too will satisfy the security theorem in the model. The FTLS will provide greater detail of the separation kernel and will provide more concrete descriptions of abstract concepts presented in the model.

C. DESCRIPTION OF SEPARATION KERNEL FTLS IN SPECWARE

1. FTLS Type Declarations

In the model many type declarations were undefined abstractions. In the FTLS, we refine the type declarations to more closely indicate how the implementation will represent those types. The complete FTLS is given in Appendix B, which is a subset of the Least Privilege Separation Kernel FTLS. In the FTLS we declare the type *Object* which represents a more concrete description of the *Resource* declared in the model. In the FTLS an object can either be a process with a unique ID, a segment in memory with a unique ID and a size, an eventcount, or a sequencer. Reed & Kanodia describe how eventcounts and sequencers can provide process synchronization of execution without the need for mutual exclusion [Ree79]. As a result, sufficient process synchronization can be

achieved within the bounds of secure information flow. It is recommended to read the work of Reed & Kanodia to have a better understanding of how eventcounts and sequencers manage information flow and process synchronization.

After the declaration of the *Object* type, we refine the concept of a *Subject* by first using the same method for declaring an exported object as in the model. There is no difference between the declaration of an exported object in the FTLS and an exported resource in the model. Since all subjects are exported objects, but not all exported objects are subjects, we defined an abstract predicate in the model called *subject?*. In the FTLS we refine the same predicate regarding a subject by stating that a subject is a process. We achieve this by constructing the predicate to be true if there exists a natural number such that the object given as input to the predicate equals the process associated with the number. The rest of the declarations in the FTLS are the same as in the model, except in the FTLS we declare type *Transform* as opposed to *Operation*. Figure 15 shows the FTLS type declarations.

```
%Types
      type Object = | Process {id: Nat}
                    | Segment {id: Nat, size: Nat}
                    | EventCT (Nat)
                    | Sequencer (Nat)
        op exported? : Object -> Boolean
      type Exp_Object = (Object | exported?)

        op subject? : Exp_Object -> Boolean
       def subject? (process) =
            ex(n: Nat) process = Process {id=n}
```

```
type Subject = (Exp_Object | subject?)


type Block = | High | Medium | Low
type Mode = | RD | WT | RW | NULL
type Effect = {subject: Subject,
               resource: Exp_Object,
               flow: Mode}
type Transform = List Effect
```

Figure 15. FTLS Type Declarations

2. FTLS Function Declarations

The function declarations in the model were only refined slightly in the FTLS; however, we added another detail regarding the system with the notion of a *CurrentAccess* table. This is similar to the current access matrix described in the Bell & Lapadula model, which is an abstraction of the hardware segment descriptors through which access to memory is controlled [Bel73]. For example, the kernel substantiates a processes right to access the memory protected by a descriptor before providing it to the process. Thereafter, the process has "current access" such that it can access memory without kernel mediation. Thus the *CurrentAccess* table represents the processor local descriptor table. We also state some basic axioms regarding the state of the effects of the system such that an effect is in the *CurrentAccess* table, only if it is in the *SR* table. Similarly we state that an effect is in the *SR* table, only if it is in the *BB* table. As in the model, the combination of the *SR* and *BB* tables represent an encoding of the security policy. The addition of the *CurrentAccess* table allows us to express properties regarding effects associated with transforms. Figure 16 shows the FTLS function declarations including the *CurrentAccess* and axiom declarations.

```
  op active? : [a] a -> Boolean
  op CurrentAccess :
           Subject * Exp_Object * Mode -> Boolean

  op BB : {(b1,b2): Block*Block |
                   active? (b1)} -> List Mode
  op SR : {(s1,r2): Subject*Exp_Object |
                   active? (s1)} -> List Mode
  op Partition : Exp_Object -> Block

axiom CurrentAccess_implies_SR is
     fa(e: Effect)
       CurrentAccess(e.subject, e.resource, e.flow) =>
         member(e.flow, SR(e.subject, e.resource))

axiom SR_implies_BB is
     fa(e: Effect)
       member(e.flow, SR(e.subject, e.resource)) =>
         member(e.flow, BB(Partition(e.subject),
                           Partition(e.resource)))
```

Figure 16. FTLS Function Declarations and CurrentAccess Axioms

Following the function declarations in the FTLS, we include the model's security policy descriptions of the *active?, BB, SR, and Partition functions* (see Figure 12), which need no further refinement. The following section discusses the declarations of the transforms.

3. FTLS Transforms

In the separation kernel model we declared there to be a type or set of *Operation*(s) and we did not further discuss any members of the set within the model. We simply left them as an abstraction and constrained properties regarding all operations. As mentioned previously, we refine the abstract type *Operation* in the FTLS to be a type *Transform.* We then enumerate all the transforms that will exist in the system. In Specware, we can define an element of a certain type by declaring an op of the desired type. We can also provide certain constraints within this declaration as well. Since a transform is a list of effects, we want to ensure that if an effect is a member of the transform then it satisfies certain properties. This is also vital to uphold the proof that the FTLS is a proper refinement of the model.

In the FTLS we declare seven transforms: *HW_Read, HW_Write, Read_Write, Ticket, Read_EventCT, Adv_EventCT,* and *Await_EventCT.* These all have certain semantics, so rather than declare them all to be of type *Transform,* we can add constraints within the declaration. This also eliminates the need for axioms that convey these constraints later in the specification. An example of such constraints can be seen in the definition of *HW_Read*. We want to ensure that if an effect is a member of *HW_Read*, then the effect's flow is of type *RD* and the effect is actually allowed based on the *CurrentAccess* table. Each transform has similar but not exact constraints. Figure 17 shows the declarations of the transforms.

```
op HW_Read :
  {t1: Transform | fa(e: Effect)
```

```
                    member(e, t1) =>
                      (e.flow = RD &&
                       CurrentAccess(e.subject,
                                     e.resource,
                                     e.flow))}

op HW_Write :
  {t2: Transform | fa(e: Effect)
                    member(e, t2) =>
                      (e.flow = WT &&
                       CurrentAccess(e.subject,
                                     e.resource,
                                     e.flow))}

op Read_Write :
  {t3: Transform | fa(e: Effect)
                    member(e, t3) =>
                      (e.flow = RW &&
                       CurrentAccess(e.subject,
                                     e.resource,
                                     e.flow))}

op Ticket :
  {t4: Transform | fa(e: Effect)
```

```
                    (member(e, t4) =>
                       (e.flow = RW &&
                        CurrentAccess(e.subject,
                                      e.resource,
                                      e.flow)))
                        &&
                        length(t4) = 1}

op Read_EventCT :
  {t5: Transform | fa(e: Effect)
                    (member(e, t5) =>
                       (e.flow = RD &&
                        CurrentAccess(e.subject,
                                      e.resource,
                                      e.flow)))
                        &&
                        length(t5) = 1}
op Adv_EventCT :
  {t6: Transform | fa(e: Effect)
                    (member(e, t6) =>
                       (e.flow = WT &&
                        CurrentAccess(e.subject,
                                      e.resource,
                                      e.flow)))
```

```
                          &&
                          length(t6) = 1}


op Await_EventCT :
  {t7: Transform | fa(e: Effect)
                     (member(e, t7) =>
                          (e.flow = RD &&
                           CurrentAccess(e.subject,
                                         e.resource,
                                         e.flow)))
                          &&
                           length(t7) = 1}
```

Figure 17. Transform Declarations

These declarations indicate specific elements of the type *Transform.* Thus the only remaining constraint we need regarding transforms is to declare that these are the only transforms that exist in the system. We also state some constraints regarding the resources within the effects of each transform. We provide these constraints as axioms discussed in the next section.

4. FTLS Axioms

The only additional semantics that need to be defined in the FTLS pertain to the transforms that have been declared. These semantics are achieved through the axioms seen in Figure 18. First we need to ensure that the transforms declared are the only transforms in the system. We do this through an axiom stating that for every entity of type transform must be one of the seven declared transforms. Another constraint

regarding transforms pertains to their effects. Each transform contains a list of effects and each effect contains a resource. We need to constrain the resources of effects associated with certain transforms based on the nature of the transform. For the *HW_Read, HW_Write,* and *Read_Write* transforms, their resources should be segments. The resources of *Ticket* should be sequencers, and the resources of the *Read_EventCT, Adv_EventCT,* and *Await_EventCT* should be an eventcount. These constraints were not included as part of the transform declarations mainly to reduce redundancy within the declarations and to provide clarity of the sets of transforms associated with each type of resource. We add these constraints through two axioms stating that if a transform is equal to *HW_Read, HW_Write,* or *Read_Write,* then for all of its effects there exists a segment that equals each effect's resource. The same is done for the eventcount axiom as seen in Figure 18.

```
axiom only_ops is
      fa(t:Transform) t = HW_Read       ||
                      t = HW_Write      ||
                      t = Read_Write    ||
                      t = Ticket        ||
                      t = Read_EventCT  ||
                      t = Adv_EventCT   ||
                      t = Await_EventCT
axiom Segment_as_Object is
      fa(e: Effect, t: Transform)
        ex(n1: Nat, n2: Nat)
         ((t = HW_Read)     ||
          (t = HW_Write)    ||
```

```
        (t = Read_Write)) &&
      member(e, t) =>
            e.resource = Segment{id=n1, size=n2}


axiom EventCT_as_Object is
      fa(e: Effect, t: Transform)
        ex(n: Nat)
         ((t = Read_EventCT)   ||
          (t = Adv_EventCT)    ||
          (t = Await_EventCT)) &&
         member(e, t) =>
                e.resource = EventCT (n)


axiom Ticket_as_Object is
      fa(e: Effect, t: Transform)
        ex(n: Nat)
        (t = Ticket) &&
         member(e, t) => e.resource = Sequencer (n)
```

Figure 18. FTLS Transform Axioms

The FTLS is now complete and now we must show that it preserves the security properties of the model. The next section describes the morphism and the associated proof obligations.

D. MORPHISM IN SPECWARE

The mapping between the model and the FTLS is done through Specware's morphism capability. Every entity in the source must map to an entity in the target in order for the morphism to be correct. Specware does a good job of pattern matching in morphisms, thus it will automatically map entities with the same names without an explicit declaration. For example, in the separation kernel model we declare a type *Subject* and we also declare a type *Subject* in the FTLS. Thus in the mapping Specware automatically maps the model *Subject* to the FTLS *Subject.* Therefore, the only explicit declarations we need to make in the morphism are the mappings from entities in the model that do not have the same name as their corresponding entities in the FTLS. We declare the morphism as a separate unit within Specware which allows us to generate proof obligations based on that unit. The morphism consists of mapping the type *Resource* to type *Object*, type *Exported_Resource* to type *Exp_Obj*, and type *Operation* to type *Transform*. All other mappings do not need explicit declaration, but could be added for clarity. The morphism will process successfully through the syntax checker and prover if all entities have been mapped appropriately such that all properties and structures are preserved. As a result of the morphism, all definitions of operations and axioms in the model become conjectures that must be proven in the FTLS.

Figure 19 shows the morphism and Figure 20 shows the associated conjectures generated by the ***show obligations*** command given within the Specware shell. We defined the morphism as the unit *Mapping*. The morphism between the model and the FTLS generates conjectures based on the definitions of *SecureEffect* and *SecureOP*, and the *operations* axiom. The *SecureEffect* and *SecureOP* conjectures appear as a result of using the *op* structure to define macro logic in the model. Normally we would not expect these functions to appear in the FTLS because they are only used to bring clarity to the definition of a "secure system". Another interesting item to note is that when we generated the proof obligations for the morphism, Specware produced two obligations with the same name, *SecureOp_def*. We defined two separate proof units in order to try to prove each obligation; however, we could not verify that we were actually disambiguating the two obligations. We are not sure as to the reason Specware generated two conjectures with the same name, but this was the only point when we encountered

this problem and we will discuss this further in Chapter V. One other bug within the morphism obligations resides in the definition of *Secure_OP_def* where it refers to *Operation*. This should refer to *Transform* (since it is being proved based on the FTLS) as seen in the *operations* conjecture. It is not known why this bug occurred, and will hopefully be addressed in the future.

```
Mapping =
morphism final_model#model ->
         final_ftls#ftls{Resource +-> Object,
                         Exported_Resource +-> Exp_Object,
                         Operation +-> Transform}
```

Figure 19. Morphism Declaration

```
conjecture operations is
    fa(e : Effect, o : Transform)
     member(e, o) =>
        e.flow = RD &&
        member(e.flow, BB(Partition(e.subject),
                          Partition(e.resource))) &&
        member(e.flow, SR(e.subject, e.resource))
        ||
        e.flow = WT &&
        member(e.flow, BB(Partition(e.subject),
```

```
                        Partition(e.resource))) &&
        member(e.flow, SR(e.subject, e.resource))
        ||
        e.flow = RW &&
        member(e.flow, BB(Partition(e.subject),
                        Partition(e.resource))) &&
        member(e.flow, SR(e.subject, e.resource))

conjecture SecureEffect_def is
    fa(effect : Effect)
        SecureEffect effect =
          (effect.flow = NULL
           || member(effect.flow,
                     BB(Partition(effect.subject),
                        Partition(effect.resource)))
              && member(effect.flow, SR(effect.subject,
                                        effect.resource)))

conjecture SecureOP_def is
    fa(nil : Operation)
     fa(operation : Operation)
        nil = operation => SecureOP operation = true
```

```
conjecture SecureOP_def is

    fa(hd : Effect, tl : List(Effect))

     fa(operation : Operation)

        ~(nil = operation) &&

        Cons(hd, tl) = operation =>

                        SecureOP operation =

                          (SecureEffect hd && SecureOP tl)
```

Figure 20. Morphism Proof Obligations

We have now completed the construction of the formal model and FTLS and shown the mechanism for the interlevel mapping. In Chapter V we discuss the analysis of Specware, within the verification paradigm, against the evaluation criteria presented in Chapter II.

V. ANALYSIS OF SPECWARE AGAINST EVALUATION CRITERIA

A. INTRODUCTION TO ANALYSIS

The previous chapter discussed our development of the separation kernel formal model, FTLS, and interlevel mapping. In this chapter we will critique the process of our experiment and present our analysis of Specware for use in the verification of high assurance systems. We based our analysis on a set of evaluation criteria. The evaluation criteria were motivated by prior work in the evaluation of verification systems by Ubhayakar [Ubh03]. Further motivation was based on requirements set forth by the Trusted Computing Exemplar (TCX) project [Irv04]. We will provide a brief overview of the evaluation criteria and then present our analysis of Specware.

B. OVERVIEW OF EVALUATION CRITERIA

In this experiment, we analyzed Specware's capabilities in eight key areas. These areas represent properties that a verification system must exhibit in order to be effective in the verification of high assurance systems. The eight properties provide the basis for our analysis of a verification system (or "tool") and are: product maturity, usability of the tool and its verification environment, theorem proving capabilities, specification language, executable specifications, multiple levels of abstraction, automatic generation of conjectures, and semantics. Product maturity relates to the age and current support of the system as well as its popularity in terms of past and current projects. Usability of the tool and its verification environment refers to how complicated the system is for users and the level of training required to use the system effectively. A tool's theorem proving capabilities must be adequate in order to provide the assurance that the specifications satisfy the requirements. Not only must the theorem prover be capable of proving complex theorems, but it also must provide intuitive dialog with the user regarding success or failure of proofs. The specification language must be able to represent the logic of security theorems, state machines and at least first order logic with quantification. The syntactic elements should be simple enough to allow for the entire development team to clearly understand the specification. Executable specifications

provide the development team with a way to test certain aspects of the systems without the introduction of further detail. The tool must be able to represent multiple levels of abstraction in order to provide a sequential progression from the abstract security policy to the concrete implementation, where each level is shown to map to the level above. The tool should also have the ability to automatically generate the full set of conjectures, based on the logic of the specification, which are required to prove the security and mapping theorems. This is necessary in order to ensure that all obligations are satisfied and that subtle obligations are not overlooked. And finally, the semantics of the verification system should be well founded such that the tool is expressive and does not prohibit efficient expression of system properties and formalisms. Our analysis will consist of describing how well Specware incorporates these concepts. Beyond these general requirements for verification of secure systems, several requirements are specific to certain systems and modeling approaches. These requirements are that the tool suite should include a non-determinism checker, a flow analyzer, and a shared resource matrix generator. We will discuss these requirements as future work and do not include them in our analysis of Specware.

C. ANALYSIS OF SPECWARE

1. Product Maturity

When choosing to use a verification tool it is important that the tool has a reasonable level of maturity. Product maturity has three measurable aspects: current product support, user training classes and tutorials, and quality of worked examples in the field. Current support is important because the specifications being produced might require support from the tool's developers in order to produce the correct semantics based on the syntactical elements of the language. Support for the interface and development environment is also critical to timely and efficient production of specifications. In addition, a more mature product might support training courses either from the vendor or a third party, which could prove valuable for new users and developers. Product maturity is also important because it implies that the tool is actually in use on other projects, which can provide useful resources, documentation, and potential collaboration. If the tool is not new and not in use on other projects, this should be a warning sign that the tool is not

very mature, not very useful, or has not succeeded in providing a beneficial alternative to other products. In addition, more robust tutorials and examples covering a larger range of common issues might exist with a popular tool that has been on the market for a longer period of time.

Specware has been under constant support and development since the mid 1990s. At the time of this writing it is in Version 4.1.3. Customer support is readily available and custom queries are handled in a timely and efficient manner. Although there does not exist a dedicated support group within Specware, it has been used on many projects to specify requirements and generate code. Documentation regarding the theoretical foundations of Specware is easily found online. Williamson mentions several projects which have used Specware [Wil01] including collaboration with Boeing, Motorola, and the NSA as noted by Widmaier [Wid00]. Specware can also provide training in the use of the tool and background in the language. The current tutorial that is provided with Specware is a good example of requirements specification and refinement capabilities proceeding to code generation; however, the tutorial does not present an impressive display of the theorem prover and its automated verification capabilities. Our analysis of the theorem prover will be presented later in this section. Overall Specware is in a very mature state and has positive customer support. It is popular for use in requirements specifications and for developing correct software.

2. Usability of Tool and Verification Environment

Within any development environment, the interface commands used to operate in the environment should be intuitive. For projects with time constraints, spending more time learning the environment implies less time being spent on development. The tool may be very powerful, but if users cannot function efficiently within the environment they may choose other tools of lesser quality, which may produce less satisfying results, but are easier to operate. A graphical user interface (GUI) is also desired to avoid command line driven operations and to provide an integrated development environment (IDE). However, the use of a GUI implies that its design is also adequate and simple. If the GUI is not intuitive, then the command line interface might be more usable. Currently Specware operates in a command line driven basis. Specware has its own shell

with unique commands used to perform certain operations on specifications. The Specware shell can operate within XEmacs or as its own application outside of XEmacs. The XEmacs environment provides some features associated with an IDE such as the Specware menu that provides shortcuts to basic commands within the Specware shell. The XEmacs environment will also provide syntax highlighting of Specware specifications reflecting the syntax given in the Specware Language Manual [Kes04]. These IDE-like features are only present in XEmacs if Specware is installed, thus these features are similar to a plug-in to XEmacs. However, XEmacs does not provide a fully functional IDE as most commands to operate within Specware must still be given from the command prompt. Commands that must be initiated from the shell include those associated with generating proof obligations as well as those for generating C and Java code. The commands to process and evaluate specifications within Specware are fairly simple and straightforward and are provided in the Specware user manual [Kes04].

When developing the separation kernel model and FTLS, the usability of the Specware shell and development environment was not inhibiting or constrictive. Overall, the Specware shell and commands were simple to understand and contained well documented support if any issues arose. The Specware shell allows the user to interact efficiently with the system in order to perform the necessary operations upon the specifications. Although it might be of interest to have a complete IDE that could be used to run Specware in the future, this could reside on top of the shell which provides the flexibility and power needed to produce and process specifications efficiently.

The verification environment should help the developer to increase efficiency of producing specifications through features such as syntax highlighting, type checking, and error checking. This is important because otherwise, specifications would be written in a simple text editor or even a on a piece of paper and it might be difficult to catch subtle errors or type inconsistencies. If the development environment provides these features, the mistakes will be caught early in the process rather than persisting until the proof is attempted. Currently Specware provides a syntax highlighting feature available through XEmacs, and when the *proc* command is issued from within the Specware shell, the processor checks the specification for type consistency as well as for common errors such as undefined parameters. The error messages are provided directly in the Specware shell

and when working in XEmacs, if an error is present, another buffer appears with the cursor placed at the line and column of the specification where error occurs. When not working in XEmacs, the error is simply output to the Specware shell. In both cases, the error messages contain the line and column position where the error occurs in the file. This provides the developer with a reference to be able to locate the error exactly. An example of an error message is given in Figure 21.

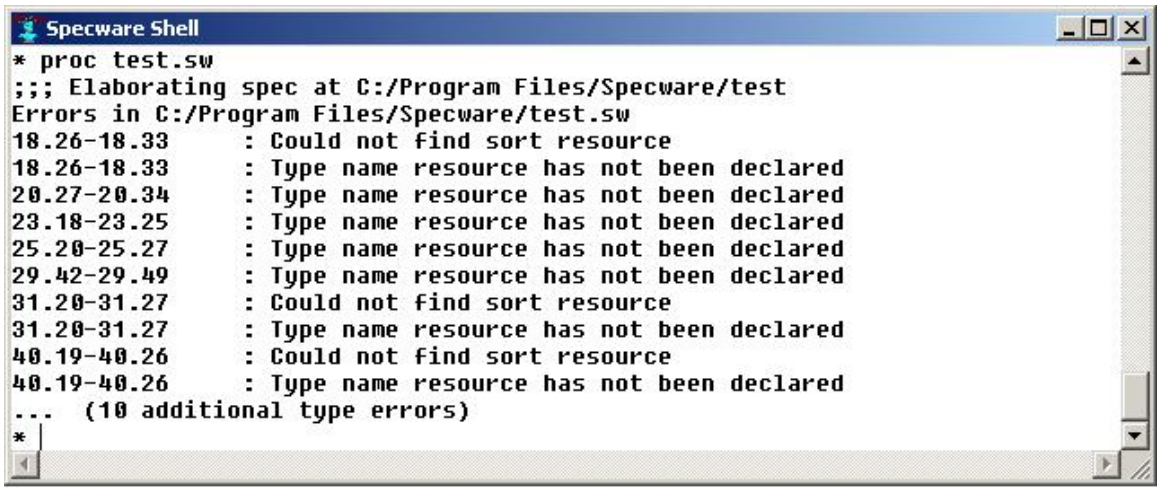

Figure 21. Specware Error Messages

Notice that in Figure 21, the first error states that on line eighteen column twenty-six through line eighteen column thirty-three, there is an error with the sort *resource* and consequently there are many more errors related to *resource*. This type of error checking is important to avoid wasting time due to syntax errors when the proofs are attempted, thus increasing the efficiency of specification writing.

3. Theorem Proving

The theorem proving capabilities of a verification tool are very important since the entire goal of the verification paradigm is to prove certain properties regarding the security policy. As noted previously there are two basic types of theorem provers, automated and interactive. Interactive theorem provers allow the user to guide the prover in proof steps whereas the automated provers simply attempt the proofs without user intervention. For small problems, model checkers can also be used, but for larger

problems model checkers cannot completely exhaust all possible states and offer little assurance. Several characteristics distinguish theorem provers. One useful characteristic is that the theorem prover should be easily integrated into the verification environment. For example, the specification processor should automatically prepare the specification to be input to the prover. This means that the user does not need to modify the specification in order to be able to invoke the prover. The theorem prover should also provide meaningful error messages when it finds errors or is unable to finish a proof. In addition, the prover should have adequate capabilities to log attempted and completed proofs. This is useful because it allows the user to trace the steps of the prover and perhaps recognize the problem if a proof has failed. It also allows the user to trace through the steps of the proof upon success in order to gain better understanding of how the proof was formulated.

Specware currently interfaces with the Snark first-order theorem prover [Kes04]. Snark is an automated theorem prover and Specware automatically pre-processes specifications to send to the Snark prover. Thus the user need not manipulate completed specifications in order to prove obligations and by issuing the *prove* command within the Specware shell, Specware will invoke Snark to prove a given unit. Once Snark has been invoked it will automatically attempt a proof of the unit and will return with a message in the Specware shell indicating whether or not the conjecture or theorem was proved or not. Snark also creates a log file of its processing on the given unit. In its raw form, the log file is not intuitive, and its comprehension was beyond the scope of this thesis. When a proof has succeeded, it is difficult to trace the log file to see what steps were taken to complete the proof. Similarly, when a proof has failed, Snark does not generate any type of helpful error messages and tracing through the log file is not possible without training in Snark. Figure 22 shows an unsuccessful proof attempt from within the Specware shell, Figure 23 shows a successful proof attempt and Figure 24 shows a snapshot of a Snark log file for the successful attempt. Note however, that the structure of the log file is the same regardless of whether or not the proof was successful. Even though the content of the log file is different based on success or failure, it remains extremely difficult to read without extensive knowledge of Snark.

```
* proc /test_oblig
…
Expanded spec file: /Program Files/Specware/Snark/..sw
Snark Log file: /Program Files/Specware/Snark/..log
Conjecture SecureOP_Obligation is NOT proved. using Snark.
*
```

Figure 22. Unsuccessful Proof Message

```
* proc /test_oblig
…
Expanded spec file: /Program Files/Specware/Snark/..sw
Snark Log file: /Program Files/Specware/Snark/..log
Theorem Secure in final_model#model is Proved! using Snark.
*
```

Figure 23. Successful Proof Message

```
    :NAME :|unary_minus_injective_on_positives|)
  (SNARK::ASSERT
     '(SNARK:ALL ((SNARK::|?n| :SORT NUMBER))
       (MES:IMPLIES (AND (>= SNARK::|?n| 0)
                         (SNARK::|Nat.posNat?|
SNARK::|?n|))
                    (=   (-   0   (-   0   SNARK::|?n|))
SNARK::|?n|)))
```

```
    :NAME :|minus_negative|)
  (SNARK::ASSERT '(= (- 0 0) 0) :NAME :|minus_zero|)
  (SNARK::ASSERT
     '(SNARK:ALL ((SNARK::|?i| :SORT NUMBER))
       (= (- 0 (- 0 SNARK::|?i|)) SNARK::|?i|))

    :NAME :|unary_minus_involution|)
  (SNARK::ASSERT '(SNARK::|Functions.bijective?| -)
    :NAME :|unary_minus_bijective|)
  (SNARK::ASSERT
     '(SNARK:ALL ((SNARK::|?i| :SORT NUMBER))
       (AND (= (+ SNARK::|?i| 0) SNARK::|?i|)
            (= (+ 0 SNARK::|?i|) SNARK::|?i|)))     :NAME
:|unary_minus_injective_on_positives|)
```

Figure 24. Snapshot of Snark Log File

Naturally it would be helpful to know why a conjecture did not prove and the only approach to figuring this out is to look into the log file given by Snark. However, the log file does not provide a clean representation of the approach the prover took and every log file is of substantial length. The log for the successful proof of the security theorem in the separation kernel model, partially shown in Figure 24, was eighty-seven pages long. However, through working with the proof obligations in Specware, we noticed that most of the other obligations did not prove in Snark even when it was intuitive that the obligation was provable. In this case, without Snark training, the options are to verify the proof by hand or declare an axiom regarding the obligation in the specification.

Overall Snark does not succeed often in proving more complicated theorems and it seems necessary to construct the axioms and definitions in such a manner as to guarantee that the prover will succeed. But this approach inhibits the developer and limits the clarity and expressiveness of specifications, which is not an acceptable option. For example, in an earlier version of our FTLS specification we included the *ex1* quantification in order to express that there exists a unique element in the set, which is perfectly acceptable in Specware's MetaSlang. When we attempted the proof of the theorem, the prover generated an error indicating that it could not handle the uniqueness quantification, resulting in a modification of the specification. This is an example of a syntactic element in Specware that cannot be handled by the theorem prover. In other cases the prover simply failed for unknown reasons. In fact, the tutorial provided by Specware contains proof obligations that do not prove in Snark. If the prover were interactive, the user might have more success in aiding the proof, and if the user were given more insight as to why the proof did not succeed, perhaps the specification could be tweaked to aid in the proof. In addition, more intuitive error messages would also help to distinguish logic errors from the inadequacy of the theorem prover.

In addition to Snark, Specware includes a simple inequality reasoning engine that attempts basic inequality proofs on conjectures before sending them to Snark. This is useful for simple proofs, but it does not provide a log file associated with the proof. A log would be useful for documentation purposes even though the developer might be able to sketch a proof by hand knowing that it can be proved using simple inequality reasoning. Note that the inequality reasoning engine is not a part of Snark, rather it was developed by Kestrel for Specware. Invoking the reasoning engine is not explicitly done by the user and the user does not need to do anything different when attempting to prove obligations.

Thus the inadequate logging and error messages of Specware's theorem prover, combined with its apparent weakness at automatically resolving logical propositions, means that seemingly simple proofs might not be proven. Currently, the best approach for theorem proving in Specware is to generate the proof obligations, prove as many obligations as possible automatically, and then verify manually those obligations that did not succeed. The proof units associated with all the obligations in our experiment are

given in Appendix D. This includes the proof units as well as output from the Specware shell indicating whether or not the proof succeeded or failed for each conjecture.

4. Specification Language

As noted earlier Specware incorporates MetaSlang as its specification language, and the formal theory behind Specware is category theory. MetaSlang is a functional language which can express powerful logical statements. Functional languages represent computation as evaluations of mathematical functions as opposed to imperative languages which use the modification of state [Wik06]. Functional language expression is highly useful when defining predicates and composing axioms and theorems regarding requirements and constraints on the system. When developing the Separation Kernel model and FTLS, we found the language powerful enough to express our requirements due to the static nature of the security policy. A limitation of a functional language is its inability to represent state-based variables, state machines, and state transitions (e.g. x' = x+1). Currently the best approach to representing state when using a functional language is through the use of the Monad construct, described comprehensively by Wadler [Wad95]. We explored the use of Monads briefly, but did not complete the analysis and leave this exploration for future work. One issue to explore is the implication that the use of Monads has on the proof obligations.

The base libraries in Specware provide elements for expressing complex logical constructs (e.g. higher order quotients [Kes04]) based on the native base logic and continue to grow. As Specware is used on more projects, the hope is that collaboration will aid in the growth of libraries, including those for addressing the state monads and state machine issues. Williamson et. al. also noted some of these same areas for growth within Specware [Wil01*]. In our experiments the language was initially challenging to understand due to lack of multiple examples or exercises and it took some time to get used to the grammar. This was due to the lack of experience within functional programming paradigm, which is important to keep in mind for developers coming from a background in imperative or procedural programming. As familiarity with the language increased it also became evident that MetaSlang provides the ability to express constructs in a very terse manner, which led to confusion between developers. For example, Figure

25 is taken from the *List.sw* library specification [Kes04]. The construct calculates the length of a list through the use of recursion. The construct is concise but not necessarily intuitive at first glance. This terseness could cause confusion and needs to be avoided in the specifications for high assurance systems. A rule of thumb when developing high assurance systems is to be as clear as possible in order to avoid confusion even if there is a more efficient manner in which to write the expression.

```
 op length          : [a]   List a -> Nat
def length l =
  case l of
     | []    -> 0
     | _::tl -> 1 + (length tl)
```

Figure 25. Example of Terse MetaSlang

The language also supports multiple layers of abstraction and includes native syntax for morphisms, as well as automatic generation of related proof obligations, allowing refinement, which is a necessity within the verification paradigm. There are some minor syntactic peculiarities associated with the grammar, but naturally the more time a developer is involved with a language, the less distracting they become. For example, the use of the "|" symbol has many different contexts such as use in the sum type, the case statement, and set comprehension (i.e. "such that").

5. Executable Specifications

The ability to execute specifications allows developers the flexibility to experiment with the semantics of the system being specified while not having to address lower level implementation details. Overall the use of executable specifications aids in

the efficiency of correctly constructing the system. Specware supports the ability to execute certain expressions in MetaSlang, particularly constructive expressions. A constructive expression is an expression where all *types* and *ops* have explicit definitions and do not include quantifications (i.e. ***ex, fa, ex1***)[Kes04]. Constructive expressions are evaluated by setting the context of the Specware shell to the *spec* term itself and then invoking the *eval* <*expression*> command. A built-in MetaSlang interpreter supports execution. Figure 26 shows an example of an executable specification and Figure 27 shows an example of a non-executable expression because it is non-constructive [Kes04].

```
spec
  def f x = 2*x+1
  def t = 6172
endspec
```

Figure 26. Executable Specification

```
spec
  def f x = 2*x+1
  op t : Nat -> Nat
endspec
```

Figure 27. Non-Executable Specification

The command *eval f t* for Figure 26 would result in 12345, whereas the command could not be executed for Figure 27. This is a nice feature but is limited to constructive expressions and therefore not all specifications can be executed. Thus to generally execute specifications, actual code would need to be generated but a specification might not be refined enough to generate the code. Specware's sister product, Planware, provides a framework for Libraries designed to provide the necessary refinements. The

fact that Specware has the ability to generate code is a beneficial feature, because it allows for quicker testing once the specification has been refined to the point of possible code generation. Even though Specware may not allow all specifications to be executed, it does allow for some execution and also provides the benefit of code generation from refined specifications.

6. Multiple Levels of Abstraction

A verification tool must be able to support multiple levels of abstraction in order to allow for the proper refinement from the security policy to the implementation. This type of refinement is achieved through incremental steps moving from more abstract concepts at a higher level to more concrete details at the lower level. Specware supports the process of refinement corroborated formally through category theoretic morphisms, colimits, and diagrams. For a comprehensive understanding of category theory, readers are encouraged to investigate other resources and texts including Pierce [Pie91] and Barr [Bar90]. Our main investigation focused on the use of the morphism. In Specware, the morphism is a structure- and property-preserving mapping between two specifications and their individual elements and operations. All axioms and definitions (i.e. the semantics of the elements and operations) in the higher (source) *spec* become conjectures in the lower (target) *spec*. Category theory provides the corollary that any theorems that are proved in the source *spec* need not be proved again in the target *spec* as long as the target *spec* is shown to uphold the axioms and definitions in the source. For example, in our mapping from the Separation Kernel model to the FTLS, the axioms and definitions in the model became conjectures in the FTLS. As long as we proved those conjectures, then the security theorem in the model will hold in the FTLS and does not need to be proved again. Thus Specware supports multiple levels of abstraction very well and ensures proper refinement between levels, whereas a verification system without this support would impose the additional requirement to verify the correctness of the mapping-theorem logic.

We mentioned in Chapter IV that we encountered a mapping problem in our initial approach to developing the model. Initially, we defined three operations, each of which was to be a prototype for a class of FTLS transforms. Thus one operation was

called a *Read* operation, one was called a *Write*, and the other a *Read_Write*, where each operation had the semantics that its effects contained the corresponding flows and the flows were allowed by the *BB* and *SR* policies. When we developed the FTLS we wanted to define more than just three operations yet we wanted to map them to our original three operations in the model. For example, we wanted to map *HW_Read*, *Read_EventCT*, and *Await_EventCT* back to the abstract *Read* function in the model. However, this type of mapping is not allowed in Specware, in which the morphism requires a one-to-one mapping. In order to satisfy the morphism, we collapsed the operations in the model into the *operations* axiom. This allowed us to preserve the semantics of the prototype transform operations and also define a legal mapping.

The only other problems we encountered when attempting the morphism between the separation kernel model and FTLS were problems regarding explicit mapping definitions. Specware developers have indicated that these problems will not exist in future releases since the problems are not logical errors and preserve the morphism properties. One problem was where we wanted to map the sum *type Mode* = | *RD* | *WT* | *EXEC* to the FTLS *type Mode* = | *READ* | *WRITE* | *EXECUTE*. Specware did not allow this mapping because the types were named differently and there was no way to explicitly map each element of the sum type in the model to another partition of the sum type in the FTLS, e.g. *RD* +-> *READ*. A similar problem with identifiers occurred when mapping types or objects of exactly the same type, but where different instantiated variables within the definition were used. Figure 28 shows an example.

```
Model:
    type Operation
    op Read : {o1: Operation |
                    fa(e: Effect) member(e, o1)}
```

```
FTLS:
    type Transformation
    op HW_Read : {t1: Transformation |
                      fa(e: Effect) member(e, t1)}
Mapping:
    morphism Model +->
                FTLS {Operation +-> Transformation,
                      Read +-> HW_Read}
```

Figure 28. Mapping Problem Example

In the example shown in Figure 28 the Specware processor throws an error due to the *t1* in the FTLS not being a *o1* as it is in the model. Thus in the FTLS we must switch the *t1* back to an *o1*. This is strange since the *o1* and *t1* are simply arbitrary and we are concerned with their use being mapped correctly and not their names.

These are minor problems that will hopefully be resolved in future releases of Specware. Overall, the refinement capabilities within Specware are powerful and provide support for multiple layers of abstraction.

7. Automatic Generation of Conjectures

When working with intricate specifications, many proof obligations can be obvious to the developers, but some obligations may be subtle. In these cases it is useful if the verification tool can automatically generate all of the conjectures to prove the soundness of the specification and mappings. Specware supports the automatic generation of conjectures when the user issues the appropriate commands from within the Specware shell. The *show obligations in* <*unit*> command displays the proof obligations that are not explicitly stated as conjectures or theorems within the specification. For example, in our Separation Kernel model, a proof obligation is not displayed by the *show*

obligations command for the security theorem because it is an explicit obligation. Once the command is issued, Specware generates a separate specification containing all of the conjectures that are not explicit. Note that, this specification is output to the Specware shell and is not created as a separate file, so it is advisable to copy the output and store the specification as a separate file for future reference. However, any obligations stated explicitly remain within the original specification. In summary, obligations that are not stated explicitly in the specification are accessed by the *show obligations* command and all other obligations remain in the specification. All proof attempts of obligations are performed using the *prove* command, followed by the unit and obligation name.

Specware also automatically generates the conjectures that must be proved for a morphism in the same manner, where the morphism is treated as a separate unit. This is valuable because developers need not worry about the obligations associated with each level of refinement as they are automatically generated by Specware. In our generation of the morphism obligations, Specware generated two conjectures with the same name, seen in Figure 20 in Chapter IV and in Appendix G. This made it difficult to attempt to prove the two conjectures as we could not verify which one was actually being attempted when we ran them in Snark. This was later determined to be a bug in Specware. However, this was the only time we ran across two conjectures being generated with the same identifier. Since the glitch we encountered revolved around unique identification, it should be easily fixed. The fact that Specware still generated a necessary conjecture is a positive aspect of the system. Thus, even though two conjectures had the same name, the conjectures were still generated and could be proved by hand in order to verify their correctness.

Overall, Specware aids in ensuring that subtle obligations are addressed throughout the development of specifications. Conjectures and theorems can be stated explicitly within the specification and can also be generated automatically by the Specware processor. The proof obligations generated by Specware for the separation kernel model, FTLS, and morphism can be found in Appendices E, F, and G respectively.

8. Semantics

The underlying logic and foundational theory behind a verification tool is important because this directly affects the expressiveness of the tool and the assurance provided. We have previously discussed that the underlying foundational theories for Specware are category theory and lambda calculus as apparent in the functional language paradigm. This foundation allows Specware to express higher order logic and refinement with minimal complexity. However, there were some challenges associated with developing our separation kernel model and FTLS. Some linguistic problems we faced were based on ambiguities in the use of certain symbols. For example, the * can either imply a product in the literal sense, such as multiplication of integers, or it could be used as a separator between input parameters in a function (e.g. *Subject*Resource -> Flow*). The use of the * in the latter example implies a cross product of the "sets" of inputs. Those not familiar with the functional language paradigm might find the "overloading" of the * symbol to be ambiguous and perhaps a different symbol would be beneficial for the sake of clarity. The major problem we faced when familiarizing ourselves with the language was determining how to express the logic that we could verbalize quite easily. This problem could easily be overcome with more robust documentation and explanation of logic within tutorial examples and specifications. It took some time to understand what certain example expressions were saying and it took time to determine the best way to express what we wanted to formalize for our specifications. Thus the foundational theory and semantics of Specware is very powerful, but there are some linguistic idiosyncrasies that must be overcome in order to utilize the full capability of Specware

D. CONCLUSION

In this chapter we have provided an analysis of Specware based on a set of evaluation criteria. We discussed the strengths of Specware as well as problems encountered regarding the development of the Separation Kernel model, FTLS, and morphism between the two specifications. The next section will discuss our conclusions regarding Specware and recommendations for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS AND FUTURE WORK

A. CONCLUSIONS OF ANALYSIS

The construction of the separation kernel formal model and FTLS within Specware allowed us to assess the role Specware can play in the verification of high assurance systems. As noted previously, a verification tool consists of a specification language and theorem proving capabilities. Overall we found that Specware is a powerful tool that has a solid foundational theory allowing it to express higher order logic in a simple and concise manner. In terms of the specification language, Specware is in a mature state and has been used in many commercial and research projects. Theorem proving support in Specware needs some improvement, as we will discuss later in this section. The Specware shell and development environment is not complicated, allowing developers to quickly become familiar and comfortable with Specware. However, Specware does not contain a fully integrated development environment. This might be a desirable feature that would increase its efficiency and usability. The Specware processor supports error checking. The error messages are terse and sometimes it is difficult to distinguish the actual error, but overall they provide sufficient context to locate the problem. Thus the Specware environment is adequate for developing formal specifications and certainly does not provide a hindrance to the development lifecycle.

The specification language, MetaSlang, incorporated by Specware is a powerful and expressive language. MetaSlang is a functional language which is valuable in terms of verification, but it has difficulty representing state based variables. Representation of state leads to the use of monads and future work will hopefully reveal what challenges this might present to verification of specifications. We will discuss potential future work regarding verification of specifications which incorporate monads later in this section.

The refinement capabilities in Specware definitely support the goals of multiple levels of abstraction needed in the verification paradigm. The morphism provides the necessary and sufficient mechanism to show that a lower level specification preserves the

security properties[1] of the level above. Specware also succeeds in its ability to express theorems and conjectures concisely in addition to the capability to automatically generate conjectures. The automatic generation of conjectures in Specware ensures that subtle obligations will not be overlooked. Specware also generates automatic conjectures when performing a morphism, which provides a means to prove the interlevel mapping. Aside from a few minor problems with the mapping syntax and semantics, Specware succeeds in providing an effective means to express multiple levels of abstraction and automatic generation of conjectures.

The theorem proving capabilities are the biggest area for improvement. Currently Specware interfaces with the automated theorem prover Snark. Snark is deficient in multiple ways including insufficient logging capabilities such that it is difficult for the user to verify the proof, or lack thereof, based on the generated log. It also struggles with proving relatively simple theorems providing no intuitive indication as to the reason for failure. The error messages are not typically helpful and will only indicate that the theorem proved or did not prove. The theorem prover drawbacks are naturally an initial deterrent when considering Specware for use in the verification paradigm, due to the fact that proofs will not be guaranteed unless produced by hand. We understand that most projects using Specware forego the actual proving of theorems. One example of not relying on the theorem prover was noted by Widmaier [Wid00]. Within the context of the verification of high assurance systems, proofs are a necessity not only to verify that the system satisfies the security policy, but also to meet desired evaluation assurance levels with respect to criteria (e.g. Common Criteria). Since Specware is not a theorem prover in and of itself, this problem can be solved relatively simply without the need to restructure the entire foundation of Specware. An interface to other theorem provers appears to be the major feature needed. To add versatility to the users and projects a generic interface would allow users to choose which prover they would like to use either based on familiarity or other requirements. For instance, some users may wish to know that the simple theorems can be proved, a job well suited for an automated prover or

[1] This thesis examined the preservation of flow properties from the perspective of subjects and their effects on exported resources. Other research has shown that a noninterference property from the perspective of traces might not be preserved by refinement unless the specifications are bi-similar (i.e. at the same level of detail) [Bib05]

model checker. Other users wish to know the formulation of the proof or provide the proof as documentation, which can be provided by an interactive theorem prover. These are examples of reasons to integrate Specware with multiple theorem provers, but the main issue is confidence in the proving capabilities. If a theorem is not proved the prover should indicate why and the path it took to the reach the point of failure. We will discuss integration with other theorem provers as future work.

In conclusion, through our analysis of Specware, we feel that Specware has the necessary components to serve as a verification system for high assurance system development, provided the improvement upon the theorem proving capabilities occurs. More research is required to understand the use of MetaSlang for state-machine formal models. We are aware that efforts are being made to improve these weaknesses and under that assumption, Specware can be very useful in the verification of high assurance systems. Specware provides a powerful specification language and is an excellent system to produce high assurance software. Furthermore, verification of high assurance systems can be enhanced with automatic code generation, although this aspect of formal system development was not investigated. Specware takes an average amount of time to become familiar with but features excellent support. It has great potential for use as a verification system in the development of high assurance systems.

B. RECOMMENDATIONS

1. Integrated Development Environment

Providing a stand alone distribution that incorporates the Specware processor underneath an integrated development environment (IDE) would be beneficial to the overall efficiency of developing formal specifications. An IDE would allow developers to create, process, and verify specifications with better organization and more graphical interaction. This would allow users to install one application to get benefits such as syntax highlighting, as opposed to first requiring XEmacs. An example of this type of IDE would be similar to Microsoft Visual Studio, or NetBeans for Java. We feel that this type of environment can also aid in the organization of Specware libraries where the IDE can manage the paths to the libraries regardless of the path under which the current specification is being developed. The IDE could also provide better organization of the

proof units associated with a project. Currently, the best way to handle the units is to place them in a separate file, but an IDE could provide a display of all obligations and an interactive dialog that would display obligations that have been attempted as well as indicate their success or failure. The IDE recommendation is merely a suggestion to provide more continuity throughout the process of development and verification within Specware.

2. Theorem Prover Integration

Integration efforts between Specware and multiple theorem provers would be valuable to aid in the confidence of proving capabilities associated with Specware as well as provide versatility to the developers. Currently Specware only interfaces with one automated theorem prover that lacks the power needed to be used in the verification of high assurance systems. This results in a lack of user confidence in Specware's theorem proving capabilities and Specware as a complete verification system. Developing the capability to interface Specware with alternate theorem provers such as PVS or Isabelle HOL would allow Specware to be used in a much broader set of verification environments. This integration would allow theorems to be proved on multiple platforms thus decreasing the amount of manual verification. For example, if a theorem could not be proved in Snark, perhaps the theorem could be proved in PVS, and if not in PVS, hopefully in HOL or even another theorem prover. Research into integrating Specware with other theorem provers would improve the versatility and capability of Specware as a verification system.

C. FUTURE WORK

1. Verification of State Representation in Specware

The use of monads to represent state in Specware is a feasible option and can be implemented. The monadic structure can be complex and implementation requires careful attention. The question for research is the effect that monads have on verification of the basic security theorem. It would be useful to develop formal specifications against a security policy that used monads to represent system state. Then we would like to attempt to prove conjectures and a basic security theorem. Creating a refinement of a

more abstract specification and being able to prove the mapping when each level uses monads to represent state would verify that Specware can incorporate verification of state-based specifications. The level of difficulty associated with such proofs would be a good point for analysis. This would prove useful for developers needing to not only incorporate state within formal specifications, but would also satisfy the need to prove security properties regarding state in the system.

2. Trusted Computing Exemplar

The Trusted Computing Exemplar (TCX) project is an ongoing research effort to develop a high assurance least privilege separation kernel [Lev04]. The model and FTLS we developed for this thesis can be enhanced to accommodate all of the requirements for the TCX separation kernel. Enhancements include incorporating a notion of initialization of the policy tables within in the model. This initialization can occur at boot up of the system or during runtime, requiring an interface which accesses multiple policies. The model would also need to specify a trusted partial ordering on the flows between blocks for the identification of "trusted subjects". The TCX project has certain requirements regarding the verification system used and we have addressed all of those requirements in our analysis of Specware except for a few, which can be met by hand or potentially with another tool. These requirements include the need for a non-determinism checker, static flow analyzer, and shared resource matrix generator. Note that these are optional requirements for which tool-based support would be desirable within the TCX project, however if these items are not available there are other avenues available to meet project objectives.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: SEPARATION KERNEL MODEL IN SPECWARE

This appendix is the Separation Kernel Model as described in Chapter IV.

```
model = spec
  %Types
     type Resource

       op exported? : Resource -> Boolean
     type Exported_Resource  = (Resource | exported?)

       op subject? : Exported_Resource -> Boolean
     type Subject = (Exported_Resource | subject?)

     type Block = | High | Medium | Low
     type Mode = | RD | WT | RW | NULL
     type Effect = {subject: Subject,
                    resource: Exported_Resource,
                    flow: Mode}
     type Operation = List Effect

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  %Definitions
       op active? : [a] a -> Boolean

  %BB and SR represent the policy tables
       op BB : {(b1,b2): Block*Block | active? (b1)}-> List Mode
       op SR : {(s1,r2): Subject*Exported_Resource | active? (s1)}-> List Mode
       op Partition : Exported_Resource -> Block

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  %Policy Description
       op SecureEffect : Effect -> Boolean
      def SecureEffect (effect) =
         (effect.flow = NULL ||
          (member(effect.flow, BB(Partition(effect.subject),
                                  Partition(effect.resource))) &&
           member(effect.flow, SR(effect.subject, effect.resource))))
```

```
      op SecureOP : Operation -> Boolean
     def SecureOP (operation) = case operation of
                                | nil -> true
                                | Cons(hd, tl) -> (SecureEffect(hd) &&
                                                  SecureOP(tl))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

   %Axiom
    axiom operations is
          fa(e: Effect, o: Operation)
              member(e,o) =>
                 (e.flow = RD &&
                  member(e.flow, BB(Partition(e.subject),
                                    Partition(e.resource))) &&
                  member(e.flow, SR(e.subject, e.resource)))
                 ||
                 (e.flow = WT &&
                  member(e.flow, BB(Partition(e.subject),
                                    Partition(e.resource))) &&
                  member(e.flow, SR(e.subject, e.resource)))
                 ||
                 (e.flow = RW &&
                  member(e.flow, BB(Partition(e.subject),
                                    Partition(e.resource))) &&
                  member(e.flow, SR(e.subject, e.resource)))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  %Theorem
  theorem Secure is
          fa(o: Operation) SecureOP(o)
endspec
```

APPENDIX B: SEPARATION KERNEL FTLS IN SPECWARE

This appendix is the Separation Kernel FTLS as described in Chapter IV.

```
ftls = spec
  %Types
     type Object = | Process {id: Nat}
                   | Segment {id: Nat, size: Nat}
                   | EventCT (Nat)
                   | Sequencer (Nat)

       op exported? : Object -> Boolean
     type Exp_Object = (Object | exported?)

       op subject? : Exp_Object -> Boolean
      def subject? (process) = ex(n: Nat) process = Process {id=n}
     type Subject = (Exp_Object | subject?)

     type Block = | High | Medium | Low
     type Mode = | RD | WT | RW | NULL
     type Effect = {subject: Subject,
                    resource: Exp_Object,
                    flow: Mode}
     type Transform = List Effect

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %Definitions
       op active? : [a] a -> Boolean

  %CurrentAccess represents the process local descriptor table
       op CurrentAccess : Subject * Exp_Object * Mode -> Boolean

  %BB and SR represent the policy tables
       op BB : {(b1,b2): Block*Block | active? (b1}} -> List Mode
       op SR : {(s1,r2): Subject*Exp_Object | active? (s1)} -> List Mode
       op Partition : Exp_Object -> Block

    axiom CurrentAccess_implies_SR is
          fa(e: Effect)
             CurrentAccess(e.subject, e.resource, e.flow) =>
                  member(e.flow, SR(e.subject, e.resource))
```

```
    axiom SR_implies_BB is
          fa(e: Effect)
             member(e.flow, SR(e.subject, e.resource)) =>
             member(e.flow, BB(Partition(e.subject), Partition(e.resource)))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  %Policy Description
       op SecureEffect : Effect -> Boolean
      def SecureEffect (effect) =
          (effect.flow = NULL ||
           (member(effect.flow, BB(Partition(effect.subject),
                                   Partition(effect.resource)))
            &&
            member(effect.flow, SR(effect.subject, effect.resource))))

       op SecureOP : Transform -> Boolean
      def SecureOP (transform) = case transform of
                                   | nil -> true
                                   | Cons(hd, tl) -> (SecureEffect(hd) &&
                                                      SecureOP(tl))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Transforms
       op HW_Read :
        {t1: Transform | fa(e: Effect) member(e, t1) =>
                                        (e.flow = RD &&
                                         CurrentAccess(e.subject,
                                                       e.resource,
                                                       e.flow))}

       op HW_Write :
        {t2: Transform | fa(e: Effect) member(e, t2) =>
                                        (e.flow = WT &&
                                         CurrentAccess(e.subject,
                                                       e.resource,
                                                       e.flow))}
```

```
op HW_Read_Write :
 {t3: Transform | fa(e: Effect) member(e, t3) =>
                                  (e.flow = RW &&
                                   CurrentAccess(e.subject,
                                                 e.resource,
                                                 e.flow))}

op Ticket :
 {t4: Transform | fa(e: Effect) (member(e, t4) =>
                                   (e.flow = RW &&
                                    CurrentAccess(e.subject,
                                                  e.resource,
                                                  e.flow)))
                                   &&
                                    length(t4) = 1}

op Read_EventCT :
 {t5: Transform | fa(e: Effect) (member(e, t5) =>
                                   (e.flow = RD &&
                                    CurrentAccess(e.subject,
                                                  e.resource,
                                                  e.flow)))
                                   &&
                                    length(t5) = 1}

op Adv_EventCT :
 {t6: Transform | fa(e: Effect) (member(e, t6) =>
                                   (e.flow = WT &&
                                    CurrentAccess(e.subject,
                                                  e.resource,
                                                  e.flow)))
                                   &&
                                    length(t6) = 1}

op Await_EventCT :
 {t7: Transform | fa(e: Effect) (member(e, t7) =>
                                   (e.flow = RD &&
                                    CurrentAccess(e.subject,
                                                  e.resource,
                                                  e.flow)))
                                   &&
                                    length(t7) = 1}
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %Axioms
    axiom only_ops is
          fa(t:Transform) t = HW_Read       ||
                          t = HW_Write      ||
                          t = HW_Read_Write ||
                          t = Ticket        ||
                          t = Read_EventCT  ||
                          t = Adv_EventCT   ||
                          t = Await_EventCT

    axiom Segment_as_Object is
          fa(e: Effect, t: Transform)
            ex(n1: Nat, n2: Nat)
             ((t = HW_Read)       ||
              (t = HW_Write)      ||
              (t = HW_Read_Write))
             &&
             member(e, t) => e.resource = Segment{id=n1, size=n2}

    axiom EventCT_as_Object is
          fa(e: Effect, t: Transform)
            ex(n: Nat)
             ((t = Read_EventCT)   ||
              (t = Adv_EventCT)    ||
              (t = Await_EventCT))
             &&
             member(e, t) => e.resource = EventCT (n)

    axiom Ticket_as_Object is
          fa(e: Effect, t: Transform)
            ex(n: Nat)
            (t = Ticket) &&
             member(e, t) => e.resource = Sequencer (n)
endspec
```

APPENDIX C: MORPHISM FROM MODEL TO FTLS

This appendix displays the morphism unit defined in Specware for the mapping between the Separation Kernel Model and FTLS as described in Chapter IV.

```
Mapping = morphism final_model#model ->
                        final_ftls#ftls{Resource +-> Object,
                                        Exported_Resource +-> Exp_Object,
                                        Operation +-> Transform}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D: SEPARATION KERNEL PROOF UNITS

This appendix provides all of the proof units for the Separation Kernel Model, FTLS, and morphism. Included with each proof unit is a snapshot of the output for each attempt indicating the success or failure of each proof.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%model_oblig1-3 did not prove, model_oblig4-9 and Security Theorem did prove
model_oblig = obligations rev_final_model#model


model_oblig1 = prove SecureEffect_Obligation in model_oblig
  %Snark Log file: H:/.../model_oblig1.log
  %model_oblig1: Conjecture SecureEffect_Obligation in model_oblig is NOT
   proved using Snark.


model_oblig2 = prove SecureEffect_Obligation0 in model_oblig
  %Snark Log file: H:/.../model_oblig2.log
  %model_oblig2: Conjecture SecureEffect_Obligation0 in model_oblig is NOT
   proved using Snark.


model_oblig3 = prove SecureOP_Obligation in model_oblig
  %Snark Log file: H:/.../model_oblig3.log
  %model_oblig3: Conjecture SecureOP_Obligation in model_oblig is NOT proved
   using Snark.


model_oblig4 = prove operations_Obligation in model_oblig
  %Snark Log file: H:/.../model_oblig4.log
  %model_oblig4: Conjecture operations_Obligation in model_oblig is Proved!
   using Snark.
```

```
model_oblig5 = prove operations_Obligation0 in model_oblig
  %Snark Log file: H:/.../model_oblig5.log
  %model_oblig5: Conjecture operations_Obligation0 in model_oblig is Proved!
   using Snark.

model_oblig6 = prove operations_Obligation1 in model_oblig
  %Snark Log file: H:/.../model_oblig6.log
  %model_oblig6: Conjecture operations_Obligation1 in model_oblig is Proved!
   using Snark.

model_oblig7 = prove operations_Obligation2 in model_oblig
  %Snark Log file: H:/.../model_oblig7.log
  %model_oblig7: Conjecture operations_Obligation2 in model_oblig is Proved!
   using Snark.

model_oblig8 = prove operations_Obligation3 in model_oblig
  %Snark Log file: H:/.../model_oblig8.log
  %model_oblig8: Conjecture operations_Obligation3 in model_oblig is Proved!
   using Snark.

model_oblig9 = prove operations_Obligation4 in model_oblig
  %Snark Log file: H:/.../model_oblig9.log
  %model_oblig9: Conjecture operations_Obligation4 in model_oblig is Proved!
   using Snark.

Model_Security_Theorem = prove Secure in model_oblig
  %Snark Log file: H:/.../Model_Security_Theorem.log
  %Model_Security_Theorem: Theorem Secure in model_oblig is Proved! using
   Snark.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%ftls_oblig1-4, ftls_oblig6 did not prove, but ftls_oblig5 did prove
ftls_oblig = obligations rev_final_ftls#ftls

ftls_oblig1 = prove CurrentAccess_implies_SR_Obligation in ftls_oblig
  %Snark Log file: H:/.../ftls_oblig1.log
  %ftls_oblig1: Conjecture CurrentAccess_implies_SR_Obligation in ftls_oblig is
   NOT proved using Snark.

ftls_oblig2 = prove SR_implies_BB_Obligation in ftls_oblig
  %Snark Log file: H:/.../ftls_oblig2.log
  %ftls_oblig2: Conjecture SR_implies_BB_Obligation in ftls_oblig is NOT proved
   using Snark.

ftls_oblig3 = prove SR_implies_BB_Obligation0 in ftls_oblig
  %Snark Log file: H:/.../ftls_oblig3.log
  %ftls_oblig3: Conjecture SR_implies_BB_Obligation0 in ftls_oblig is NOT
   proved using Snark.

ftls_oblig4 = prove SecureEffect_Obligation in ftls_oblig
  %Snark Log file: H:/.../ftls_oblig4.log
  %ftls_oblig4: Conjecture SecureEffect_Obligation in ftls_oblig is NOT proved
   using Snark.

ftls_oblig5 = prove SecureEffect_Obligation0 in ftls_oblig
  %Snark Log file: H:/.../ftls_oblig5.log
  %ftls_oblig5: Conjecture SecureEffect_Obligation0 in ftls_oblig is Proved!
   using Snark.
```

```
ftls_oblig6 = prove SecureOP_Obligation in ftls_oblig
  %Snark Log file: H:/.../ftls_oblig6.log
  %ftls_oblig6: Conjecture SecureOP_Obligation in ftls_oblig is NOT proved
   using Snark.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%mapping_oblig1&2 did not prove
%mapping_oblig3-5 proved using simple inequality reasoning

mapping_oblig = obligations rev_final_ftls#Mapping

mapping_oblig1 = prove operations in mapping_oblig
  %Snark Log file: H:/.../mapping_oblig2.log
  %mapping_oblig2: Conjecture operations in mapping_oblig is NOT proved using
   Snark.

mapping_oblig2 = prove SecureEffect_def in mapping_oblig
  %Snark Log file: H:/.../mapping_oblig3.log
  %mapping_oblig3: Axiom SecureEffect_def in mapping_oblig is Proved! using
   simple inequality reasoning.

mapping_oblig3 = prove SecureOP_def in mapping_oblig
  %Snark Log file: H:/.../mapping_oblig4.log
  %mapping_oblig4: Axiom SecureOP_def in mapping_oblig is Proved! using simple
   inequality reasoning.

mapping_oblig4 = prove SecureOP_def in mapping_oblig
  %Snark Log file: H:/.../mapping_oblig5.log
  %mapping_oblig5: Axiom SecureOP_def in mapping_oblig is Proved! using simple
   inequality reasoning.
```

APPENDIX E: SEPARATION KERNEL MODEL PROOF OBLIGATIONS

This appendix displays the output of the automatically generated proof obligations for the Separation Kernel Model.

```
import /Library/Base/WFO

conjecture SecureEffect_Obligation is

    fa(effect : Effect)

      ~(effect.flow = NULL) => active?(Partition(effect.subject))


conjecture SecureEffect_Obligation0 is

    fa(effect : Effect)

     ~(effect.flow = NULL)

     && member(effect.flow, BB(Partition(effect.subject),

                               Partition(effect.resource)))

     => active?(effect.subject)


conjecture SecureOP_Obligation is

    ex(pred : List(Effect) * List(Effect) -> Boolean)

     WFO.wfo pred

     && (fa(operation : Operation, tl : List(Effect), hd : Effect)

          (operation = Cons(hd, tl) && SecureEffect hd => pred(tl, operation)))


 conjecture operations_Obligation is

    fa(e : Effect, o : Operation)

     member(e, o) && e.flow = RD => active?(Partition(e.subject))
```

```
conjecture operations_Obligation0 is
    fa(e : Effect, o : Operation)
     member(e, o)
     && e.flow = RD
     && member(e.flow, BB(Partition(e.subject), Partition(e.resource)))
     => active?(e.subject)

conjecture operations_Obligation1 is
    fa(e : Effect, o : Operation)
     member(e, o)
     && ~(e.flow = RD
           && member(e.flow, BB(Partition(e.subject), Partition(e.resource)))
           && member(e.flow, SR(e.subject, e.resource)))
     && e.flow = WT
     => active?(Partition(e.subject))

conjecture operations_Obligation2 is
    fa(e : Effect, o : Operation)
     member(e, o)
     && ~(e.flow = RD
           && member(e.flow, BB(Partition(e.subject), Partition(e.resource)))
           && member(e.flow, SR(e.subject, e.resource)))
     && e.flow = WT
     && member(e.flow, BB(Partition(e.subject), Partition(e.resource)))
     => active?(e.subject)
```

```
conjecture operations_Obligation3 is
    fa(e : Effect, o : Operation)
     member(e, o)
     && ~(e.flow = RD
           && member(e.flow, BB(Partition(e.subject), Partition(e.resource)))
           && member(e.flow, SR(e.subject, e.resource)))
     && ~(e.flow = WT
           && member(e.flow, BB(Partition(e.subject), Partition(e.resource)))
           && member(e.flow, SR(e.subject, e.resource)))
     && e.flow = RW
     => active?(Partition(e.subject))

conjecture operations_Obligation4 is
    fa(e : Effect, o : Operation)
     member(e, o)
     && ~(e.flow = RD
           && member(e.flow, BB(Partition(e.subject), Partition(e.resource)))
           && member(e.flow, SR(e.subject, e.resource)))
     && ~(e.flow = WT
           && member(e.flow, BB(Partition(e.subject), Partition(e.resource)))
           && member(e.flow, SR(e.subject, e.resource)))
     && e.flow = RW
     && member(e.flow, BB(Partition(e.subject), Partition(e.resource)))
     => active?(e.subject)
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX F: SEPARATION KERNEL FTLS PROOF OBLIGATIONS

This appendix displays the output of the automatically generated proof obligations for the Separation Kernel FTLS.

```
import /Library/Base/WFO

conjecture CurrentAccess_implies_SR_Obligation is
    fa(e : Effect)
     CurrentAccess(e.subject, e.resource, e.flow)
     => active?(e.subject)


conjecture SR_implies_BB_Obligation is
    fa(e : Effect)
     active?(e.subject)


conjecture SR_implies_BB_Obligation0 is
    fa(e : Effect)
     member(e.flow, SR(e.subject, e.resource))
     => active?(Partition(e.subject))


conjecture SecureEffect_Obligation is
    fa(effect : Effect)
     ~(effect.flow = NULL)
     => active?(Partition(effect.subject))
```

```
conjecture SecureEffect_Obligation0 is
    fa(effect : Effect)
     ~(effect.flow = NULL)
     && member(effect.flow, BB(Partition(effect.subject),
                               Partition(effect.resource)))
     => active?(effect.subject)


conjecture SecureOP_Obligation is
    ex(pred : List(Effect) * List(Effect) -> Boolean)
     WFO.wfo pred
     && (fa(transform : Transform, tl : List(Effect), hd : Effect)
          (transform = Cons(hd, tl)
           && SecureEffect hd
            => pred(tl, transform)))
```

APPENDIX G: SEPARATION KERNEL MORPHISM PROOF OBLIGATIONS

This appendix displays the output of the automatically generated proof obligations for the Separation Kernel morphism.

```
import /H:/.../rev_final_ftls#ftls

conjecture operations is

    fa(e : Effect, o : Transform)

     member(e, o)

     => e.flow = RD

        && member(e.flow, BB(Partition(e.subject), Partition(e.resource)))

        && member(e.flow, SR(e.subject, e.resource))

     || e.flow = WT

        && member(e.flow, BB(Partition(e.subject), Partition(e.resource)))

        && member(e.flow, SR(e.subject, e.resource))

     || e.flow = RW

        && member(e.flow, BB(Partition(e.subject), Partition(e.resource)))

        && member(e.flow, SR(e.subject, e.resource))


conjecture SecureEffect_def is

    fa(effect : Effect)

     SecureEffect effect =

       (effect.flow = NULL

        || member(effect.flow, BB(Partition(effect.subject),

                                  Partition(effect.resource)))

           && member(effect.flow, SR(effect.subject, effect.resource)))


conjecture SecureOP_def is

    fa(nil : Operation)

     fa(operation : Operation)
```

```
      nil = operation => SecureOP operation = true



conjecture SecureOP_def is
    fa(hd : Effect, tl : List(Effect))
     fa(operation : Operation)
      ~(nil = operation)
      && Cons(hd, tl) = operation
      => SecureOP operation = (SecureEffect hd && SecureOP tl)
```

LIST OF REFERENCES

[Bar90] Barr, M., and Wells, C. *Category Theory for Computing Science*, Prentice Hall. 1990.

[Bel73] Bell, D.E., and LaPadula, L.J. *Secure computer systems: mathematical foundations and model*. M74-244, The MITRE Corp., Bedford, Mass., May 1973.

[Bib05] Bibighaus, D. L. *Applying Doubly Labeled Transition Systems to the Refinement Paradox.* Doctoral Dissertation, Naval Postgraduate School, Monterey, CA, September 2005.

[Com06] _. *Common Criteria Documentation.* <http://www.commoncriteriaportal.org/> May 2006.

[Irv04] Irvine, C. E., Levin, T. E., Nguyen, T. D., and Dinolt, G. W. *The Trusted Computing Exemplar Project.* Proc. 2004 IEEE Systems, Man and Cybernetics Information Assurance Workshop, West Point, NY, pp. 109-115. June 2004.

[Kes04] Kestrel. *Specware Documentation*. <http://www.specware.org/doc.html>. September 2004.

[Kre99] Kremman, T.W., Martin, W.B., and Taylor, F.S. *An Avenue for High Confidence Applications in the 21st Century.* National Security Agency. February 1999.

[Lan81] Landwehr, Carl E. *Formal Models for Computer Security*. Naval Research Laborotory. September 1981.

[Lev04] Levin, T.E., Irvine, C.E., and Nguyen T.D. *A Least Privilege Mode for Static Separation Kernels.* Naval Postgraduate School, Monterey, CA,, Technical Report NPS-CS-05-003. October 2004.

[Mar00] Martin, W.B., White, P.D., and Vanfleet, W.M. *Government, Industry, and Academia: Teaming to design High Confidence Information Security Applications.* Proc. Third Workshop on Formal Methods in Software Practice**,** pp. 37-47. 2000.

[McD01] McDonald, J. and Anton, J. *SPECWARE Producing Software Correct by Construction.* Kestrel Institute. March 2001.

[NSA87] NSA. *FTLS Must Accurately Describe TCB Operations (C1-CI-01-87).* Last retrieved from http://niap.nist.gov/cc-scheme/PUBLIC/0237.html on June 5, 2006. March 1987.

[Par92] Paramax Systems Corporation. *FDM User Guide*. June 1992.

[Pav03] Pavlovic, D., and Smith, D.R. *Software Development by Refinement.* Lecture Notes in Computer Science, Vol. 2757, pp. 267 – 286. January 2003.

[Pav04] Pavlovic, D., Pepper P., and Smith, D. *Colimits for Concurrent Collectors.* Lecture Notes in Computer Science, Vol. 2772, pp. 568 – 597. January 2004.

[Pie91] Pierce, B. *Basic Category Theory for Computer Scientists.* MIT Press. 1991.

[Ree79] Reed, D.P., and Kanodia, R.K. *Synchronization with Eventcounts and Sequencers.* Comm. ACM. Vol. 2, No. 2. February 1979.

[Sri95] Srinivas, Y.V., and Jüllig, R. *Specware: Formal Support for Composing Software*. In Mathematics of Program Construction. July 1995.

[Sri96] Srinivas, Y.V., and McDonald, J.L. *The Architecture of Specware, A Formal Software Development System*. Kestrel Institute Technical Report KES.U.96.7. August 1996.

[Ster91] Sterne, Daniel. *On the Buzzword "Security Policy"*. Proc. 1991 IEEE Computer Society Symposium on Research in Security and Privacy, pp. 219-230, May 1991.

[Ubh03] Ubhayakar, Sonali. *Evaluation of Program Specification and Verification Systems.* Master's Thesis, Naval Postgraduate School. June 2003.

[Wad95] Wadler, Philip. *Monads for Functional Programming*. In Advanced Functional Programming, volume 925 of LNCS, pp. 24-52. May 1995.

[Wid00] Widmaier, J.C., Smidts, C., and Xin Huang. *Producing more reliable software: mature software engineering process vs. state-of-the-art technology?* Proc. 2000 International Conference on Software Engineering, pp. 88-93. June 2000.

[Wik06] Wikipedia. *Functional Programming.* Retrieved from <http://en.wikipedia.org/wiki/Functional_programming> on May 9, 2006.

[Wil01] Williamson, Keith. *Systems Synthesis: Towards a new paradigm and discipline for knowledge, software, and system development and maintenance.* Mathematics and Computing Technology Boeing Phantom Works. March 2001.

[Wil01*] Wiliamson, K., Healy, M., Barker, R. *Industrial Applications of Software Synthesis via Category Theory – Case Studies Using Specware.* Journal of Automated Software Engineering, Vol. 8, pp. 7-30. 2001.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. John Anton
   Kestrel Technology, LLC
   Los Altos, CA

4. Hugo A. Badillo
   NSA
   Fort Meade, MD

5. George Bieber
   OSD
   Washington, DC

6. RADM Joseph Burns
   Fort George Meade, MD

7. John Campbell
   National Security Agency
   Fort Meade, MD

8. Alessandro Coglio
   Kestrel Technology, LLC
   Los Altos, CA

9. Deborah Cooper
   DC Associates, LLC
   Roslyn, VA

10. CDR Daniel L. Currie
    PMW 161
    San Diego, CA

11. Louise Davidson
    National Geospatial Agency
    Bethesda, MD

12. Steve Davis
NRO
Chantilly, VA

13. Dr. Robert DeCloss
Northwest Nazarene University
Nampa, ID

14. Vincent J. DiMaria
National Security Agency
Fort Meade, MD

15. CDR James Downey
NAVSEA
Washington, DC

16. Dr. Diana Gant
National Science Foundation

17. Jennifer Guild
SPAWAR
Charleston, SC

18. Richard Hale
DISA
Falls Church, VA

19. CDR Scott D. Heller
SPAWAR
San Diego, CA

20. Wiley Jones
OSD
Washington, DC

21. Russell Jones
N641
Arlington, VA

22. Tim Kremann
NSA
Ft. Meade, MD

23. David Ladd
Microsoft Corporation
Redmond, WA

24. Dr. Carl Landwehr
DTO
Fort George T. Meade, MD

25. Steve LaFountain
NSA
Fort Meade, MD

26. Dr. Greg Larson
IDA
Alexandria, VA

27. Dr. Karl Levitt
NSF
Arlington, VA

28. Dr. Vic Maconachy
NSA
Fort Meade, MD

29. Doug Maughan
Department of Homeland Security
Washington, DC

30. Dr. John Monastra
Aerospace Corporation
Chantilly, VA

31. John Mildner
SPAWAR
Charleston, SC

32. Dr. Barry Myers
Northwest Nazarene University
Nampa, ID

33. Mark T. Powell
Federal Aviation Administration
Washington, DC

34. Jim Roberts
Central Intelligence Agency
Reston, VA

35. Jon Rolf
NSA
Fort Meade, MD

36. Keith Schwalm
Good Harbor Consulting, LLC
Washington, DC

37. Charles Sherupski
Sherassoc
Round Hill, VA

38. Ken Shotting
NSA
Fort Meade, MD

39. CDR Wayne Slocum
SPAWAR
San Diego, CA

40. Dr. Ralph Wachter
ONR
Arlington, VA

41. David Wirth
N641
Arlington, VA

42. CAPT Robert Zellmann
CNO Staff N614
Arlington, VA

43. Dr. Cynthia E. Irvine
Naval Postgraduate School
Monterey, CA

44. Thuy D. Nguyen
Naval Postgraduate School
Monterey, CA

45. Timothy E. Levin
Naval Postgraduate School
Monterey, CA

46. Daniel DeCloss
Affiliation (SFS students: Civilian, Naval Postgraduate School)
Monterey, CA